Master's thesis

Master's Programme in Computer Science

Managing Network Delay for Browser Multiplayer Games

Kimmo Lepola

November 4, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Markku Kojo, Valtteri Niemi

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Kimmo Lepola

Työn nimi — Arbetets titel — Title

Managing Network Delay for Browser Multiplayer Games

Ohjaajat — Handledare — Supervisors

Markku Kojo, Valtteri Niemi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 4, 2020	81 pages

Tiivistelmä — Referat — Abstract

Latency is one of the key performance elements affecting the quality of experience (QoE) in computer games. Latency in the context of games can be defined as the time between the user input and the result on the screen. In order for the QoE to be satisfactory the game needs to be able to react fast enough to player input. In networked multiplayer games, latency is composed of network delay and local delays. Some major sources of network delay are queuing delay and head-of-line (HOL) blocking delay. Network delay in the Internet can be even in the order of seconds.

In this thesis we discuss what feasible networking solutions exist for browser multiplayer games. We conduct a literature study to analyze the Differentiated Services architecture, some salient Active Queue Management (AQM) algorithms (RED, PIE, CoDel and FQ-CoDel), the Explicit Congestion Notification (ECN) concept and network protocols for web browser (WebSocket, QUIC and WebRTC).

RED, PIE and CoDel as single-queue implementations would be sub-optimal for providing low latency to game traffic. FQ-CoDel is a multi-queue AQM and provides flow separation that is able to prevent queue-building bulk transfers from notably hampering latency-sensitive flows.

WebRTC Data-Channel seems promising for games since it can be used for sending arbitrary application data and it can avoid HOL blocking. None of the network protocols, however, provide completely satisfactory support for the transport needs of multiplayer games: WebRTC is not designed for client-server connections, QUIC is not designed for traffic patterns typical for multiplayer games and WebSocket would require parallel connections to mitigate the effects of HOL blocking.

ACM Computing Classification System (CCS)
CCS → Networks → Network protocols
CCS → Networks → Network algorithms → Data path algorithms
CCS → Applied computing → Computers in other domains → Personal computers and PC applications → Computer games

Avainsanat — Nyckelord — Keywords

multiplayer games, browser games, network delay, network protocols, AQM, ECN, DiffServ

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking and Services subprogramme

Contents

1 Introduction

Video game industry is predicted to produce 196 billion (196 thousand million) US dollars in revenue by the year 2022 [Web19]. The industry goes through a constant change, in technology and in its business models. A growing share of games is released as free-to-play (F2P) while cloud gaming is also on the rise. In 2019, already 82% ($87.1B) of the revenue came using the F2P model [Lu20]. Majority of it was from mobile games. In 2020, the amount of people who are playing video games is estimated to be more than 2.5 billion [Wij19].

Different game genres are plenty. A blog post [Vin18] about video game genres lists a total of 49 different ones. A web article [Jon19] ranking the most popular video game genres mentions Massive Multiplayer Online (MMO) games, Sports games and First-Person Shooter (FPS) games, to name a few.

Different games have different requirements for the system. These requirements include the performance of the local system (for example, computer hardware) and, in case of networked multiplayer games, also the performance of the network.

Network performance can be measured in terms of throughput, latency, jitter and packet drop rate [Ken02, pp. 64–66], [Tec15]. Throughput is the amount of data per time unit going through the network from the sender to the receiver, latency is the time it takes for the data to travel from the sender to the receiver, jitter is the variation in latency and packet drop rate is the amount of packets lost per time unit during the transfer.

The requirements vary from game to game. Some games have more strict requirements for latency than others. Some require higher throughput than others. Some games can handle packet drops better than others. Cloud gaming, for example, requires a lot higher throughput than a non-cloud regular game. Cloud gaming is a technology where the game is rendered at a remote server and a video stream is sent to the player client. In a regular game, only game state information needs to be transferred from the server to the client whereas in cloud gaming, where no game logic is executed at the client, everything that is drawn onto the screen needs to be transferred from the server to the client.

Browser games are video games that use the web browser as a platform. Browser games do not need installation but instead they can be launched immediately by simply entering

an address into the address bar of the web browser. This lowers the threshold for gaming. However, the web browser as a platform has some limitations regarding the use of certain underlying technical solutions. Most prominently, these are multi-threading and network protocols. The multi-threading and network protocol options that are commonly available for a desktop application are not directly available for an application running in the web browser. The limited possibilities in the web browser pose an extra challenge on the game design and can result in solutions that unnecessarily increase the latency experienced in the game.

Latency is one of the key performance factors affecting the quality of experience (QoE) in games. Latency in the context of games can be defined as the time between the user input and the result on the screen. The game needs to be able to react fast enough to player input in order to produce outcome that feels intuitive for the player [Sav+14].

The more fast-paced the game is the more stricter latency requirements it often has [DWW05; RSR08; CC06]. The most demanding game genre in terms of latency requirements can be considered to be the FPS games [SS15]. Studies [CC06; DWW05; Qua+04] show that in FPS games network round-trip times under 100 ms would usually be preferred.

In networked multiplayer games, latency is composed of network delay and local delay [RP15]. Network delay is the time it takes for a message to travel from the sender to the receiver. Local delay is the time between a user input, such as a mouse click, and the resulting change on the screen. Local delay does not include the possible network delay in-between.

Local delay consists of input device latency, rendering delay, screen refresh rate and monitor response time. Between the input and the rendering of the result of the input, information may be sent to a server and back in order to receive a confirmation of the result, such as hit to a target. Thus, the overall latency, from input to result on the screen, may involve both the local delay and network delay.

Various sources of delay contribute to network delay [Bri+16]. Some major sources of network delay in the context of multiplayer games are signal propagation delay, queuing delay, packet loss recovery delay and head-of-line (HOL) blocking delay. Signal propagation delay is the time it takes for a bit to travel from a network node to another. Queuing delay is the time a packet is at a queue inside a network node waiting for its turn to be forwarded. Packet loss recovery delay is the delay caused by packet retransmission. HOL blocking is a situation where packet ordering requirements prevent a packet from being

delivered from the receiver buffer to the upper layer or the application.

Network delay in the Internet can be in the order of hundreds or even thousands of milliseconds [GN11]. In addition to this, local delay in real-world gaming scenarios can be around 20-240 ms [Ivk+15]. Both the network delay and local delay can thus be substantial contributors for the overall latency. In this thesis, however, we leave out local delay and focus solely on the network delay.

Multiplayer game network traffic usually consists of frequently sent small packets carrying user input and game state information [CC06; SS15; CI12]. (This does not apply to cloud gaming, which we leave outside of this thesis.) The possible strict latency expectations of the packets pose a challenge on networking solutions. If the packets have to queue for a long time behind competing traffic in the routers and switches on the path, they may easily miss their deadline set by the game. Also, if the game uses a transport protocol that suffers from HOL blocking, reordered or lost packets will cause additional delays. On top of this, signal propagation causes delay relative to the travel distance, adding to the overall delay.

Queuing in network devices is managed with queuing and scheduling algorithms. Also, various policies are in use in the Internet to control how traffic is treated while in the queues. These policies are often realized with the Differentiated Services (DiffServ) architecture [BBC06] that provides a concept for classifying traffic and treating packets differently based on their traffic class.

From the delay types, our main focus in this thesis will be on queuing delays and HOL blocking delays. We consider Active Queue Management (AQM) algorithms [BF15] and the DiffServ architecture for managing the queuing delays. Regarding HOL blocking, we consider communication protocols that can avoid HOL blocking and are available for web browsers. We also consider Explicit Congestion Notification (ECN) [FRB01] in reducing packet loss recovery delays and HOL blocking.

The common transport protocols used in the Internet are Transmission Control Protocol (TCP) [Pos81] and User Datagram Protocol (UDP) [Pos80] [Ant15, pp. 263–264]. TCP guarantees reliable in-order delivery and, therefore, suffers from HOL blocking. UDP does not make such guarantees and thus avoids HOL blocking. For this reason, games that require low latency usually use UDP as their transport protocol [MR16].

Web browsers, however, do not expose raw UDP or TCP sockets. Instead, UDP and TCP sockets are accessed via higher layer protocols or APIs. For the purposes of this thesis

we examine three such protocols/APIs: WebSocket [MF11], QUIC [IT20] and WebRTC [Alv17].

WebSocket is a transport protocol that uses TCP as a substrate. We include it in this study to give perspective on the other two options. QUIC is a transport protocol that uses UDP as a substrate. In browsers, QUIC is accessible via Hypertext Transfer Protocol HTTP/3 [Bis20]. WebRTC utilizes multiple protocols for which it uses UDP underneath. WebRTC is designed for peer-to-peer communications and in that context foremostly for video calls.

For this thesis, we formulate our research question as: "What feasible networking solutions exist for browser multiplayer games?" We conduct a literature study to analyze some salient AQM algorithms and differentiated packet treatment concepts, the ECN mechanism and some network protocols and consider how they would be suitable for supporting multiplayer game traffic.

The remainder of this thesis is organized as follows. In Chapter 2 we discuss network delay in the context of multiplayer games. In Chapter 3 we consider the support a network can provide for latency-sensitive traffic in the form of AQM, differentiated packet treatment and ECN. In Chapter 4 we examine the WebSocket, QUIC and WebRTC protocols and in Chapter 5 we conclude the thesis.

2 Network Delay in Multiplayer Games

In this chapter we will first give an overview on networked multiplayer game architectures, their components, traffic characteristics and the specifics of using web browser as a platform for games. Then we describe the various network delay types and consider the effect they have on multiplayer games.

2.1 Multiplayer Games

A game, as defined by Salen and Zimmerman [SZ03], is "a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome". Games played on PC or game consoles, or on other systems comprising an input device and video display, are called video games [Esp05].

Games can be single or multiplayer [ACB06, pp. 5–6]. Single player video games involve only one human player playing a session of the game. The game may have some sort of an artificial intelligence affecting the play and possibly acting as an opponent for the human player. Multiplayer games have two or more human players usually affecting each other's play. They may play against each other or play cooperatively.

Multiplayer game architectures

Multiplayer video games can be played on the same or on different machines [ACB06, pp. 5–6]. Games played on the same machine may implement a split screen where half of the screen is the view of the first player and another half is the view of the second player. The players may also view their avatars from the same screen if the type of the game makes it possible. This is common, for example, in some sports video games. One option also is to make players take turns when playing on the same machine. A player may complete or fail a level and then the other player may try.

Multiplayer video games played on different machines are called networked multiplayer games[†] [ACB06, pp. 5–6]. They require sending information between the participating

[†]For the rest of this thesis, the term "multiplayer game" is used in the meaning of "networked multiplayer game".

machines over a network. Possible architectures for the communication include a) client-server, b) peer-to-peer and c) peer-to-peer, client-server hybrid [ACB06, pp. 15–16].

In a client-server architecture, a typical way to structure the communication is as follows [Sil15, p. 17]. Client listens for local player input and sends the input to the server. The server processes the input and sends updated game state to appropriate clients. Upon receiving the message from the server the client updates the local game state to represent the state at the server. The clients do not send messages directly to each other but instead the information goes through the server. The server holds an authoritative state of the game.

In a peer-to-peer architecture there is no server to hold an authoritative state or mediate messages between clients [ACB06, p. 15]. Instead, the player machines, the peers, connect directly to each other. None of the peers has more control over the state of the game than others. A peer communicates its game input or state directly to other peers and there is no authority in between to decide the validity of the information. A peer-to-peer, client-server hybrid on the other hand is an architecture where a server in-between can ensure the validity of the client messages that are essential for a consistent state of the game but other non-essential information such as voice communication between players is sent peer-to-peer [ACB06, p. 16].

A client-server architecture creates a single point of failure where the server not being able to keep up with the client messages may cause the quality of experience to decrease for all the players [ACB06, p. 16]. Peer-to-peer architecture, on the other hand, may present challenges in keeping a consistent state of the game especially if a peer is purposely sending incorrect information, that is, cheating [RFP08, p. 588]. In commercial use, client-server architecture is the most popular [ACB06, p. 16].

An architecture of multiple servers is also possible [ACB06, p. 17]. Servers can operate in a peer-to-peer manner to distribute the load of one server to multiple servers. Servers themselves can also have a client-server-like architecture where one server is authoritative and the other servers communicate with it and distribute the load. Each server then only deals with a subset of the actual clients which reduces the processing and network capacity required.

Game loop

A central component of a game program is a game loop [Mad18; Nys14]. Game loop iterates through the functions that the game needs to perform in order to draw a new frame to the screen. This mechanism is a notable difference in games compared to traditional applications such as word or image processing where the program only executes functions after user input. In games the game loop continues running regardless of user input (until an end condition or exit). The loop needs to repeat as often as new content is drawn to the screen [TBN06]. For example, if the frame rate is 30 frames per second the game loop needs to repeat approximately every 33 ms.

The game loop consists of three major steps: gathering possible user input and incoming network messages, running the game simulation and rendering the game [TBN06]. In the first step the program gathers the input such as keystrokes or mouse events from the user. It also gathers the messages that have arrived through the network from the game server or from peers, depending on the architecture. Next, the program sends information of the input to the server or peers and runs the simulation to advance the game. This part involves all the logic required in updating the game state, such as physical simulation, game logic, artificial intelligence, particle systems, etc. The received input and network messages are used in calculating a proposition for the next game state. After this, the game entity interactions and collisions are resolved and the next game state is ready for rendering. The rendering part involves calculating lighting and texture states in order to create renderable representations of the game entities. This phase produces data structures that can be used outputting data to the video and sound devices.

The information the clients and the server send to each other, and the frequency of this communication, is a game design decision. A client could send data of the player avatar's position and orientation, in addition to sending only information of input events. The server could send only information considered relevant such as positions and orientations of the objects near the current player, instead of sending the whole state of the game to all the clients every time.

The sending of the user input does not need to happen at every game loop. For example, if the user is repeatedly pressing a button, the program can gather the inputs and send them grouped in a single message. Similarly, game state data incoming from the server does not necessarily come at the same frequency as the game loops. Multiple game state messages or none may arrive during a loop, depending on the game design and network

conditions.

The amount of data and the frequency of sending it, is a trade-off between game state consistency and network traffic [Fie15a]. The longer the time between sending a game state update the more time the receiver has to deviate from that state. The shorter the time between sending updates the more network traffic is generated.

In principle, many of the tasks in the game loop could be run in parallel [TBN06]. Input gathering, network message gathering, running the game simulation and rendering the game can be independent operations. Input and network message gathering produce information that the game simulation uses. Game simulation runs independently of whether input or network messages were received or not. Input and network messages are gathered regardless of what happens with game simulation or rendering. The game simulation produces information that the rendering phase uses. Rendering can or should be done whether the simulation phase updates or not. If the simulation did not update before the next render, the render should use the previous information available and execute on time. The rendering phase can utilize interpolation or extrapolation to produce the new render when no new information from the simulation is available. All these phases can be thus independent. For example, the game simulation can loop at a different frequency than the rendering. While the rendering loops every 33 ms the simulation can loop every 50 ms (assuming rendering component interpolation or extrapolation). Also some of the sub-tasks inside the phases are independent. This allows taking advantage of parallel computing and achieving possible performance gains.

Traffic characteristics in networked multiplayer games

Latency affects the user QoE (Quality of Experience). Different games have different thresholds for levels of latency before the QoE perceivably starts to degrade. Games can be divided into three categories based on the player perspective of the game world and the interaction model in the game [CC06]: omnipresent, third-person avatar and first-person avatar. Omnipresent games are such where the player views the game world mainly from above and controls a set of resources. In third-person avatar games the player controls a single character and the view follows this character. In first-person avatar games the player controls a single character and the perspective is from the eyes of the character.

Games such as First-Person Shooters (FPS) belong to the first-person avatar category. Massively Multiplayer Online Role-Playing Games (MMORPG) usually belong to the

third-person avatar category but they may also use other camera perspectives [Oli18]. Real-Time Strategy (RTS) games belong to the omnipresent category [XI15, p. 1218]. Tolerable latencies are around 80-100 ms for FPS games, around 120-500 ms for MMORPG games and around 1000 ms for RTS games [DWW05; RSR08; CC06]. In RTS games the network latency is not as relevant for QoE as it is in FPS and MMORPG games because the focus of the play in RTS games is more in strategy than in interaction [Cla05].

Networked multiplayer game traffic typically consists of frequently sent small packets [CC06; SS15]. Among the genres, FPS games usually have the smallest packets and highest sending rates. A survey [CI12, p. 242] on traffic characteristics of multiplayer games found that the packet payload sizes in FPS games were 5-300 bytes and packet inter-arrival times were 10-200 ms. MMORPG genre had packet payload sizes of about 1-600 bytes and packet inter-arrival times of about 0-3000 ms. RTS games had about 9-60 byte of packet payload size and 0-300 ms packet inter-arrival times. The maximum transmission unit (MTU) in the Internet is commonly 1500 bytes [Cox20; Hor84]. Game data packets should fit in this limit easily and not become fragmented.

Web browser as a platform

The web browser as a platform has some limitations regarding parallel computing. JavaScript code is executed in a single thread [Sil15, p. 65]. It does provide Web Workers that spawn parallel threads but data between a Web Worker and the main thread is copied rather than shared which may have negative performance implications [MDN20a; MDN20e]. However, JavaScript's SharedArrayBuffer [MDN20d] allows sharing data between threads. Also, the Transferable interface enables transferring objects with a zero-copy operation and can be utilized in transferring data between the main thread and a Web Worker [MDN20e].

For the game loop, JavaScript's window.requestAnimationFrame() method can be used [MDN20f]. This method takes a callback and requests the browser to invoke it before the next repaint. Within this function the program can run the game simulation with the input and network messages received thus far and provide the content to be rendered in the repaint. The method in the callback takes an argument which indicates a current time and it can be used for delta timing, that is, to run the simulation at the same speed regardless of the frame rate.

In addition to parallel computing the web browser as a platform has some limitations on

Delays along transmission paths	Signal propagation delay
	Medium acquisition delays
	Serialization delay
	Link error recovery delays
	Switching/forwarding delay
	Queuing delay
Structural delays	Sub-optimal routes/path
	Name resolution
	Content placement
	Service architecture
Interaction between endpoints	Transport initialization
	Secure session initialization
	Packet loss recovery delays
	Message aggregation delays
Delays related to link capacities	Insufficient capacity
	Redundant information
	Under-utilized capacity
	Collateral damage
Intra-end-host delays	Transport protocol stack buffering
	Transport head-of-line (HOL) blocking
	Operating system delays

Figure 2.1: Network delay types [Bri+16, p. 2151]

network protocol use. A game developer might want to use a bare UDP and build some own custom in-game protocol on top of it in order to have more control over how the network packets are handled at the endpoints. In non-browser games, that is, desktop games, this is possible, but in browser games this is not possible. The browser does not expose raw UDP or TCP sockets for the application. Hence, some higher level protocols such as Web Socket, QUIC or WebRTC need to be used.

2.2 Network Delay Types

Multiple elements contribute to the overall delay in communication between endpoints. To consider the reasons for delay we can use the categorization of [Bri+16, p. 2151] which divides network delays into five main types: (I) delays along transmission paths, (II) structural delays, (III) delays related to interaction between endpoints, (IV) delays related to link capacities and (V) intra-end-host delays. Each type has its own sub-types, as is illustrated in Figure 2.1. In the following we will consider these delay types from their relevant parts.

Delays along transmission paths

Delays along transmission paths are delays that happen when a packet travels in the network from an endpoint to another [Bri+16, p. 2164]. They are made of signal prop-

agation delay, medium acquisition delay, serialization delay, link error recovery delays, switching/forwarding delay and queuing delay.

Signal propagation delay depends on the distance between the sender and receiver and the medium the signal travels on [Bri+16, p. 2164]. Light travels in a vacuum about 300 mm in a nanosecond and in optical fiber about 200 mm in a nanosecond. Straighter routes between sender and receiver and development of media where signal travels faster are the means to reduce signal propagation delay.

Medium acquisition delays arise when the access to a medium needs to be shared with multiple devices [Bri+16, pp. 2164–2165]. Many techniques exist in granting the access, such as predefined time slots or random access. In these techniques channel throughput is often prioritized over latency but latency optimizations are also possible.

The process of switching/forwarding consists of multiple steps: deserializing a packet, possibly buffering it at the device input, examining its header, passing it through the switching fabric to the output corresponding to its destination, possibly buffering the packet and then serializing it [Bri+16, pp. 2165–2167]. Serialization delay consists of network devices serializing or deserializing a packet frame in order to send it to the wire or read it from the wire. Besides buffering and serialization delays, switching/forwarding delay is affected by the access speed to the forwarding table entries and complexity of rules while examining the header, and the time it takes for the packet to traverse the switching fabric from input to output.

Link error recovery delays depend on the protection the link provides against errors [Bri+16, p. 2166]. The channel coding has an effect on the serialization delay and also on the link error recovery delays. More robust channel coding against errors can reduce the link error recovery delays but increase the serialization delays because of possibly longer codes. In case corrupted data is found, link-level retransmission may be performed. On the other hand, some forwarding methods reduce the serialization delay by forwarding the packet as soon as the header is read. This, however, prevents discovering errors until the end of the packet. In this case, corrupted packets get forwarded.

Queuing delays are created at network devices when packets wait to be forwarded [Bri+16, p. 2167]. Network devices buffer incoming packets in order to accommodate for bursty traffic and to provide high link utilization. Queues build up when packets arrive faster than they can be forwarded. Larger buffers can fit longer queues and absorb bigger bursts of traffic but at the same time they can lead to longer delays. This has become a problem in the Internet where middlebox vendors installing overly large buffers to prevent packet

drops has created excessive queuing in the devices [GN11, p. 4]. This phenomenon is known as the bufferbloat. In general, queuing delays are the biggest contributing factor to delays along end-to-end path in the Internet [Bri+16, p. 2167].

The following seven topics deal with reducing the queuing delays: flow and circuit scheduling, reducing MAC buffering, smaller network buffers, transport-based queue control, traffic shaping and policing, packet scheduling, and queue management [Bri+16, p. 2167].

(1) Flow and circuit scheduling comprises techniques that seek to directly connect network device's input and output ports and thus avoid queuing [MR08; Bri+16].

(2) Reducing MAC buffering is a subject of considering what buffering is necessary at the MAC layer or below and what buffering could be done at the IP layer, in order to reduce complexity of buffering on multiple layers and achieve shorter delays [Bri+16, pp. 2167–2168].

(3) Smaller network buffers can reduce queuing delays [Bri+16, p. 2168]. Buffer size is a tradeoff between latency, utilization and packet loss.

(4) Transport based queue control methods try to accomplish low queuing delays with burstiness reduction [FF96; Kob06] or by detecting congestion in its early stage [Ali+10; HR13; Bri+16]. Some methods also try coupling the congestion control of flows that originate from the same end-host and share a bottleneck link [IWG13; WNG11; Bri+16]. Sharing a link causes competition and coupling the congestion control can reduce the latency of the flows.

(5) Traffic shaping limits transmission rates and traffic policing drops packets that exceed a specified rate [Bri+16, p. 2169]. Examples of traffic shaping techniques are a leaky bucket algorithm [Tur86] and a token bucket algorithm [TW10, p. 408]. Limiting transmission rates is a source of delay itself but on the other hand it can reduce congestion and the delay caused by the congestion by preventing a burst of packets being forwarded and possibly congesting a link in the path [Bri+16, p. 2169]. Traffic shaping is used widely by ISPs [KD11].

(6) Packet scheduling can provide fairness or prioritization to traffic [Bri+16, p. 2168]. For example, latency-sensitive flows can be prioritized over bulk transfers. A scheduler controls which packet to send when a buffer has multiple queues.

(7) Queue management controls the queues in buffers and decides which packet to drop when the queue becomes too long [Bri+16, p. 2169]. Passive techniques include drop tail and drop front. Active Queue Management (AQM) techniques include algorithms such as

RED, [FJ93] PIE [Pan+17] and CoDel [Nic+18]. AQM is an important tool in fighting the bufferbloat problem, with the algorithms focused on preventing an excessive queue build-up.

Structural delays

Structural delays relate to physical locations of network nodes and the paths between them. Structural delays are created by sub-optimal routes, name resolution, content placement and service architecture [Bri+16, pp. 2151–2157].

The path that a network packet travels from an endpoint to another is affected by Internet Service Providers' (ISP) agreements with each other, which are typically made on economic grounds. The path that an ISP forwards a packet may not be the fastest one but instead a cheapest one for the ISP. Thus, in terms of network delay, packets may often travel on sub-optimal routes.

Name resolution is required when using a domain name to find out an IP address [Moc87]. For example, a client connecting to a web server is a typical case when name resolution may be needed. Name resolution is done by using Domain Name Service (DNS) servers. A local DNS server may have cached the address which would make the name resolution relatively fast: a round-trip time (RTT) to the server [Bri+16, pp. 2153–2154]. In case of a cache miss the resolution is slower since it requires possibly traveling up and down the DNS hierarchy until the address can be found.

Due to speed of light and technological limitations on how fast a bit can travel on a medium, content placement is crucial [Bri+16, pp. 2154–2156]. The closer the content is to the user the faster it can potentially be accessed. Various kinds of caching can be implemented to bring the content closer. However, if the content is generated real time, caching static content is for no use. Network proxies that can hold a migrated application state could instead be used to bring dynamic content closer to users. Also, client caching content that a server has pushed can reduce delay, if receiving the content would otherwise require the client making a request. On the other hand, if the application usage follows a predictable pattern it may be possible for the client to request data beforehand. If the data itself is predictable, such as next position for an object following the laws of physics, the application can attempt approximating it without needing to receive it at all and thus hiding the effects of network delay.

Choices in service architecture also affect the delay [Bri+16, pp. 2156–2157]. Cloud services

or peer-to-peer (P2P) solutions can be utilized to bring the content closer for multiple users in different geographical areas. Cloud services allow spawning of server instances or caches near users, based on demand. P2P solutions can omit servers completely and host content directly on users but it requires carefully chosen topology in order to provide latency benefits. Some delay reductions can also be achieved by architectural choices of replacing chains of middleboxes with a server that runs the same functions in a single memory address space. This reduces the amounts of serialization and deserialization required.

Interaction between endpoints

Interaction between endpoints involves delays caused by transport initialization, secure session initialization, packet loss recovery and message aggregation [Bri+16, pp. 2157–2164].

Initializing transport and a secure session requires some protocol handshake operations. These handshakes may take multiple RTTs. Reducing the need for the operations to be sequential can speed up the process since the endpoint does not have to wait a full RTT for each operation to complete before proceeding to the next one. Persistent sessions and multiplexing data over a session also reduce the need for handshakes.

Packet loss recovery delays are created when packets are lost and they need to be retransmitted [Bri+16, pp. 2162–2163]. Packets may be lost due to link errors that corrupt data or because of congestion that requires packets to be dropped. Transport protocols that offer reliable delivery need to detect and retransmit the lost packets. Transport protocols that do not offer reliable delivery do not need to detect and retransmit lost packets but instead handling the situation is left to the application or a higher layer protocol. Some transport protocols may provide partial reliability in which case packets are retransmitted only in certain conditions [Ram+04]. In addition to retransmission at the transport layer the link layer may also implement its own retransmissions [Bri+16, p. 2162].

Transport protocols may use acknowledgments (ACKs) from the receiver to detect packet loss [BPA09; Ste07; IS20]. When receiving a packet the receiver sends an ACK to the sender. When receiving a certain amount of ACKs that fail to acknowledge an expected packet number the packet may be deemed lost. There is thus some delay between the actual loss of the packet and the detection of the loss. If no ACKs at all are received for a certain period of time even though one or more packets have been sent, a retransmission time-out (RTO) may occur. This is to detect a packet loss in the case there is only one or

a small number of packets in flight at a time or in case multiple consecutive packets get lost, or the loss happens at the end of the transmission. RTO incurs further delays in the packet loss recovery process [Bri+16, p. 2162].

Another way of managing packet loss is to use redundancy [Bri+16, p. 2162]. The sender can send one or more copies of the same packet to avoid the need for retransmission in case any of the copies go through. Forward error correction (FEC) is a technique of coding data redundantly into multiple packets and being able to retrieve the original data even though some of the packets get lost. Redundancy requires more capacity but reduces possible delays caused by retransmissions.

If the application can cope with missing data, it may not need retransmissions or redundancy but instead it can try to conceal the gaps, in order to improve user experience [Bri+16, p. 2162]. Concealing can be attempted by approximating the missing content based on the received content.

Many transport protocols infer congestion from packet loss [BPA09; Ste07; IS20]. They increase their transmission rate until packets start to get dropped and by this way find out the current conditions. Another way to receive information about congestion is to use Explicit Congestion Notification (ECN) [Bri+16; FRB01]. ECN enables detecting congestion without the need for packet drop. Router sets an ECN-CE (Congestion Experienced) flag in a packet IP header for incipient congestion, receiver echoes this to the sender and the sender reduces its transmission rate to avoid packet drop. This way, packet losses and associated delays may be reduced.

Some protocols try to aggregate messages in order to improve bandwidth efficiency [Bri+16, p. 2163]. This causes message aggregation delays. When about to send a packet the transport protocol may wait for a short time in anticipation of possibly receiving more data from the application to coalesce into the same packet. An example of this is the Nagle's algorithm [TW10, p. 566]. Also, sending of acknowledgments may be delayed [Bri+16, p. 2163]. The transport protocol may send an acknowledgment for every two full-sized segments only or try to piggyback it with a data segment. If neither of these conditions become fulfilled a timer will finally trigger the sending of an acknowledgment.

Delays related to link capacities

Delays related to link capacities are created by insufficient capacity or by the way the capacity is used and shared among competing traffic flows [Bri+16, p. 2172]. When

capacity is scarce, queues and congestion can build up and flows may suffer collateral damage, that is, flows may experience packet drops and delays due to competition on the link. On the other hand, under-utilizing existing capacity can unnecessarily increase flow completion times.

Capacity can be increased by upgrading the physical link interfaces to support higher transmission rates and by implementing technologies that utilize parallel links or multiple paths to distribute the load [Bri+16, p. 2172]. The need for capacity can be reduced by avoiding sending of redundant information [Bri+16, pp. 2172–2173]. For example, by using header compression or by replacing redundant unicast transmissions with a single multicast transmission, when possible. Capacity utilization can be improved with better optimized congestion control algorithms that can quickly adapt to varying traffic situations [Bri+16, pp. 2173–2175].

Intra-end-host delays

Intra-end-host delays are formed based on the design and architecture of operating systems, applications and hardware [Bri+16, pp. 2176–2177]. Central parts for delays are buffering and how information is passed between components. Data sent over the network needs to be buffered at the protocol stack of the host, in between the application and the network. Depending on traffic characteristics and the protocols used, some flows may require more buffering than others. Too small buffers can cause under-utilization. Too big buffers can allow inefficiently long queues that can add unnecessary delay.

Regardless of queue handling techniques, ordering requirements can create extra delay at the receiving endpoints [Bri+16, p. 2177]. If the transport protocol requires packets to be delivered in order then packets following a lost packet can not be delivered to the application before the lost one is retransmitted and delivered. Likewise, if a packet is not lost but reordered in the network, the following packets have to wait at the receiver buffer until the preceding packet arrives. These are occurences of head-of-line (HOL) blocking. Typically, stream-based transports, such as TCP, require the socket API to deliver data sequentially (in order) whereas datagram-based transports, such as UDP, can deliver data unordered. Some multiplexing protocols, such as QUIC, provide ordered delivery within a stream and unordered delivery between streams.

2.3 Delay Effect

The fundamental effect of network delay is that a game input registered at one place can not be registered in another place at the same instant. It takes some time for the information to travel from A to B. Network delay can thus create some discrepancy on the game state consistency between players. It is not possible to both (a) update the local game state immediately based on user input and (b) maintain consistency across all hosts all time. The design of the game has to choose between these two and make a compromise between some level of waiting and some level of inconsistency.

For example, let us consider that at time 0 the Player A presses a key to jump. At time 1 the information of the keypress has reached the server. The server sends the information to all clients. At time 2 the information reaches Player B. There are two prominent options here. Should the game at Player A execute the jump at time 0 or at time 2? If the jump is executed instantly at the keypress at time 0 then the states at Player A and Player B will differ. If the jump is executed at the same time on both players then Player A has to wait.

To manage the game state consistency various alternative methods can be implemented. Some salient concepts are the lockstep method, the snapshot interpolation method and the snapshot extrapolation method [Fie14a; Fie14b; Fie15b; ACB06].

In a lockstep method, the game waits to gather input from all players for each step and only advances when all the input for a step is received. The game is as fast as the slowest client, that is, the client who has the longest network delay determines the speed at which the game can advance for all the players.

In a snapshot interpolation method, the server receives inputs from the clients, calculates a new state based on the inputs and sends the new state (state 1) to the clients. The clients then interpolate* from their old state (state 0) to this new state (state 1). After reaching the new state, two alternative things can happen: (a) If a following new state (state 2) is received by the time this current state (state 1) is reached, the client continues to interpolate towards the new state (from state 1 to state 2); (b) If the new state (state 2) is not received by the time the client reaches the current state (state 1), the client stops the game from advancing because there is nowhere to interpolate to. When the new state (state 2) is finally received, the client once again interpolates to it from its current state.

*In interpolation, a value is approximated based on preceding and following values.

Conceptually, in snapshot interpolation, the client can be thought to be always behind time of the authoritative state.

In a snapshot extrapolation method, the server receives inputs from the clients, calculates a new state based on the inputs and sends the new state to the clients. Different from the snapshot interpolation method, the clients do not wait for the new authoritative state to arrive from the server but instead advance forward, by using extrapolation*, which, in this context, is also known as dead-reckoning. When the authoritative state update arrives from the server the clients have already advanced past it but they use it as a basis for the extrapolation in order to constantly correct their deviation from it. Hence, conceptually, in snapshot extrapolation, the client can be thought to be always ahead of time of the authoritative state.

Snapshot interpolation and extrapolation can be combined in the game's design. When the client receives information of the new authoritative state, the client can continue by interpolating towards the new state. When the client reaches the authoritative state but has not yet received information of the next authoritative state, for example due to network conditions delaying the delivery of the state message, the client can continue by extrapolating. Once the next authoritative state is received, the client can stop extrapolating and continue with interpolation towards the new state.

In the following, we will further examine the different network delay types in the context of multiplayer games.

Delays along transmission paths

Speed of light sets a hard limit on how low a signal propagation delay can be. It takes about 113 ms for light to travel the distance from London to Sydney and back, for example. This is a significant amount of time considering the preferred latency for First-Person Shooter games is below 100 ms. With current technology the light in fiber can travel about 200 000 000 meters per second [Gri13, p. 6]. This would make about 170 ms from London to Sydney and back if no other delays existed on the path. In practice, it takes about 300 ms [Won20]. Geographically optimized content placement and client/peer selection for a game session is thus crucial.

Medium acquisition techniques focus largely on throughput and are thus often favorable to bulk transmissions over delay-sensitive game flows. Serialization and switching/forwarding

*In extrapolation, a value is approximated based on preceding values.

delay on the other hand is usually proportional to the size of a packet [Bri+16, p. 2165], which is a favorable property for thin game flows compared to traffic with bigger packets. However, the smaller the packet the bigger the overhead is from packet headers.

Recovering from an error can often be done faster on the link layer than on the transport layer [Bri+16, p. 2166]. However, the game might be using a non-reliable transport protocol in an attempt to minimize latency, and not implement an error recovery scheme at all but instead handle packet loss with in-game delay-compensation methods. In this case, when the game can handle packet loss and is trying to achieve minimal latency, even the link-layer error recovery might be considered an unwelcome source of delay.

Queuing forms a major source of delay for flows in the Internet. At worst, bufferbloat can cause delays on the order of multiple seconds [GN11]. Reducing buffer sizes and thus reducing maximum queue lengths could benefit latency-sensitive traffic. However, regardless of buffer size, the question still remains on how different flows with different expectations, for latency and throughput, for example, are serviced together in the queuing and scheduling schemes of the network devices.

Structural delays

Sub-optimal routes are a hindrance to any traffic that would benefit from low latency. They add delay to all packets on the path and can reduce the quality of experience (QoE) for any real-time application. Delay caused by name resolution, on the other hand, affects only the resolution part and after that has no effect on the latency experienced in the use of an application. A game client might only need to do a name resolution once when firing up the game but otherwise use the known address for all later communication in the game.

Content placement is an essential question for real-time applications. A game which has dynamic content generated real-time does not particularly benefit from caching. The server and clients or peers producing the dynamic content should ideally be located close to each other. In some cases the client or server could predict an upcoming need for certain static content such as map data. Using the client caching method the server could push the data upfront into the cache of the client. Alternatively, using client prediction the client could request the data beforehand.

Client prediction for dynamic content is a common delay compensation method in real-time games. Client prediction methods can use previously received data and client input

to approximate future game state. Also, as another delay compensation method, the client can locally delay self-initiated actions in order to give time for the action to be communicated across the network.

The chosen service architecture has some fundamental implications on the game design and traffic patterns. Client-server architecture has the server as a central node where all traffic travels through it forming a single point of failure whereas peer-to-peer architecture distributes this load. Peer-to-peer communication has a potential for smaller delay compared to client-server since messages can travel directly from player to player without server in-between. Both architectures involve the question of how to choose the latency-wise optimal group of nodes (peers/clients and server(s)) when setting up a game session.

Interaction between endpoints

Transport initialization and secure session initialization are operations that only need to be done once in the lifetime of a connection. This event taking exceedingly long can somewhat affect the player QoE. However, it is separate from the actual gameplay where low latency is crucial.

Packet loss recovery, instead, can create substantial delays during the play. If this is not taken into account in the design of the game some appreciable jerkiness can occur. Transport protocols that guarantee reliable in-order delivery and thus suffer from HOL blocking can create sporadic delays in the order of some hundred milliseconds, depending on the RTT and the rate the client or server or the peers send updates. Also, an RTO may happen, which incurs even more delay. A host sending at a low rate, such as a client sending only player inputs, could be more susceptible for RTO than a host sending at a high rate, due to the possible lack of continuous stream of packets to generate ACKs.

When using transport protocols that do not guarantee in-order delivery the application needs to tolerate the packets possibly missing or arriving reordered. Games often use timestamps in the packets to correctly construct ordering and timing of events [Gre18, p. 1101] [Sta13, p. 197] and in-game delay-compensation methods such as extrapolation to cover for data that are missing or arrive after deadline [Sta13, p. 190]. Forward error correction or other redundancy might be used to mitigate losses although data can quickly become stale and in some cases the deadline does not allow waiting for the next packet for corrections.

Message aggregation saves capacity but is detrimental to real-time flows due to the extra

delay incurred. Game state updates and user inputs form small independent messages and there is usually no benefit in waiting and coalescing them.

Delays related to link capacities

Transport protocol choice can affect the use of the available link capacity. For example, some protocols have more overhead than others due to larger headers and thus require more capacity for transporting the same payload. Also, different protocols implement different kinds of congestion and flow control mechanisms and their flow characteristics and reaction to congestion can therefore differ, which, in turn, can result in differences in the use of the available capacity.

The choice of protocols, however, may be limited. The environment, such as the web browser, may only allow the use of certain protocols. Web browsers, for instance, do not expose raw UDP or TCP sockets but instead the web application developer has to settle for the available higher layer protocols, such as WebSocket, QUIC or WebRTC.

To save capacity, reducing redundant information may be attempted. Networked multiplayer games may involve sending game state updates with identical content to each of the participants. This is often done in unicast even though multicast as a concept would be a perfect match for this task and could save capacity. However, the support for multicast from ISPs is limited and in practice it is not available for most applications [Dio+00].

Another way to reduce redundancy is compression of data. Sending game data in as compact form as possible saves capacity. It can also reduce the probability of the packet getting fragmented or dropped due to exceeded path maximum transmission unit (MTU) size. Smaller packets usually have shorter serialization and forwarding delay as well. In addition to payload data compression, the packet headers may be compressed automatically by the protocol, as is, for example, with QUIC.

Intra-end-host delays

Along with inefficiently large buffers and excessive queuing, HOL blocking can be a major source for delays. Protocols that use TCP as a substrate suffer from HOL blocking, whereas protocols that use UDP as a substrate can avoid HOL blocking. When using UDP and requiring a reliable ordering of packets, the ordering needs to be implemented at an upper layer or by the application since UDP does not provide any ordering. Ordering

Time	Server receive	Server send	Lost in network	Client receive	Client send	Client application receive
0		p0				
25				p0	p1ACK	p0
50	p0ACK	p1				
75			p1			
100		p2				
125				p2	p1ACK	
150	p1ACK	p3				
175				p3	p1ACK	
200	p1ACK	p4				
225				p4	p1ACK	
250	p1ACK	p5, p1				
275				p5, p1	p6ACK	p1, p2, p3, p4, p5

Figure 2.2: Head-of-line blocking

requirements, however, incur HOL blocking. If no reliable ordering is implemented when using UDP, HOL blocking can be avoided.

To consider the effect of HOL blocking, let us assume the following. A game server sends updates to clients 20 times per second, that is, every 50 ms. The RTT is 50 ms. A reliable in-order transport protocol (such as, TCP) is used, and three duplicate ACK's denote a packet is lost. For simplicity, let us assume no delay variation, no congestion window and no intra-end-host delays. Figure 2.2 illustrates this case.

At time zero ms (t0) the server sends a packet zero (p0). At t25 the client receives p0, sends back an ACK (p1ACK) and the client network socket delivers p0 to the application. At t50 the server receives p1ACK and also sends p1, but p1 gets dropped in the network. Now the client network socket does not deliver the following received data in the packets (that is, p2, p3, p4...) to the application until it receives p1. The server sends p2 at t100, p3 at t150 and p4 at t200 which the client receives and sends a duplicate ACK per each. The server receives the third duplicate ACK at t250, considers p1 lost and retransmits it along with p5. The client receives p1 at t275 and delivers it to the application along with p2, p3, p4 and p5. Thus, at t25 the application receives p0, and 250 ms later, at t275, it receives the following packets p1...p5. The packets p0 and p5 are on time and packets p1...p4 are late with p1 being late 200 ms and p2 being late 150 ms.

The same case with non-ordered transport protocol is illustrated in Figure 2.3. This would be as follows. The server sends p0 at t0 and the client receives it at t25 and delivers it to the application. The server sends p1 at t50 and it gets dropped in the network. The server sends p2 at t100 and the client receives it at t125 and delivers it to the application. The server sends p3...p5 at t150...t250 and the client receives them at t175...t275 and delivers them to the application. Thus, all the packets arrive on time, except p1 which is lost.

Time	Server receive	Server send	Lost in network	Client receive	Client send	Client application receive
0		p0				
25				p0		p0
50		p1				
75			p1			
100		p2				
125				p2		p2
150		p3				
175				p3		p3
200		p4				
225				p4		p4
250		p5				
275				p5		p5

Figure 2.3: No head-of-line blocking

In the described example, the loss of one packet caused a 200 ms extra gap with the reliable in-order transport. With the non-ordered transport, the loss of one packet caused only a 50 ms extra gap. The sending rate affects the length of the gaps and the time it takes to produce the three duplicate ACKs. The RTT affects the time it takes for the third duplicate ACK and the retransmission to travel. Thus, along with the ordering requirements, the send rate and RTT affect the eventual delay that results from the packet loss.

2.4 In-game delay-compensation

The latencies in the interactions and possible state discrepancies are innate problems with network delay and need to be addressed in the game's design. Games often implement various delay compensation (also called latency compensation or latency hiding) methods to hide the effects of latency. Possible methods include local lag, local perception filter, remote lag, dead-reckoning and time warp.

Local lag is a technique where the execution of the user inputs are intentionally delayed [CCG19; PW02; Sav+14; XW13]. Some games may require that a player initiated action, such as character jumping after player key press, is executed at the same time or nearly at same time on all the participating hosts (players' machines), which the local lag method may make possible. The inputs are sent immediately to the network but their execution at the local game is delayed. This narrows, or possibly eliminates, the gap between the time the user action is carried out at the local machine and on the other players' machines.

Local perception filter is a method where an action is carried out at slightly different

speeds between clients to compensate for latency [SNH04; SRR98]. For example, a player may initiate some action which triggers an animation on all participating player machines. Due to network delay, it takes time to communicate this action to the participants and the animation will start later at remote player's machine. In order for the remote player to catch up, the animation can be played faster at the remote player and slower on the initiating player.

Remote lag is a technique where the playout of the state updates are intentionally delayed [Fie15b; Sav+14; SS15]. This method can reduce the variance in the experienced latency by utilizing playout (or jitter) buffers. A playout buffer receives state update messages from the server (or from a peer) and releases them for execution at a steady rate. This technique inserts additional delay to network messages and may thus reduce consistency between players because the longer the delay the more the game states have time for diverging from each other. However, the benefit is that the consistency level can stay constant due to steady rate of updates. Steady consistency can make the game more predictable to play [Sav+14, p. 1346].

Fast-paced games often use dead-reckoning to proceed to following game steps and do not stop and wait for authoritative game state to arrive from the server [Ber01; Fie15b; SS15], [Sav+14, pp. 1346–1347]. The local view will be an approximation of the authoritative state. The approximation is based on the previous states of the objects, such as their speed, acceleration and direction among other things.

When the authoritative state is received the local view has typically diverged from it. There are a few options on how to handle this situation [SG14; Sav+14]. One option is to let it be and not correct it if the difference is small enough. Another one is to make a correction where the objects are instantly set to their correct state, e.g. snap object to its correct position. Third one is to use interpolation and linearly smooth out the difference, e.g. move an object towards its correct place at a constant speed. Fourth one is the same as the third but with an exponential smoothing, e.g. first accelerating and then slowing down, in an effort to make the correction less noticeable.

Savery and Graham [SG14] analyze the perceivability of corrections to moving objects after they have diverged from their correct positions. They find that it is essential where the locus of attention of the player is. If the correction is done where the player is focusing at, around 50% of change relative to its normal speed may go undetected. If the correction is done outside of the area of the attention, significantly faster correction is possible. They also find that a gradually changing speed is better than a sudden change or warping.

Smooth correction is found to be better than warping even if the correction time is as short as 250 ms.

In fast-paced games such as First-Person Shooters a locally initiated action, such as the player taking a shot, may be executed locally right away without waiting for the information to reach the other players (or the server). In case of a client-server architecture, the server typically provides an authoritative state of the game and decides the outcomes of consistency affecting actions, such as if a shot hit a player or not. However, by the time the information of the action travels from the player to the server the game has already advanced forward and the server can not calculate the outcome of the action based on the current state at the server. For example, a player takes a shot at time 0 ms and by the time the information of the shot reaches the server the game at the server is already at time 100 ms. The server needs to game to be at the time the action was performed (at time 0 ms and not at time 100 ms) in order to decide the correct outcome. In this case, a method called time warp can be utilized for reviewing and playing out the passed event.

With the time warp method, player inputs are saved in a buffer and the game state can be rolled back to a certain moment and the inputs played again [MCB08; Ber01; LC18; SS15]. When the server receives the information of the action it checks the time stamp, rolls back the state to that point in time, executes the action and replays the game to see what happens. This requires that the server keeps a buffer of the past states and inputs to be able to roll the replay. After the replay the server can make an authoritative decision of the outcome.

The decision can be made from different perspectives [Ber01; LC18; Sav+14; SS15]. It may be the perspective of the player who performed the action, it may be the perspective of a player who the action affects, or it may be the perspective of the game state at the server at the time the input was received.

To consider the problematic of different perspectives, let us assume a First-Person Shooter game where a Player A has a RTT of 100 ms to the game server and a Player B has a RTT of 100 ms as well. For the sake of simplicity let us assume no delay variation and no intra-end-host processing delay. At time 0 ms Player A shoots at Player B. At time 100 ms the server receives this input and sends it to Player B. At time 200 ms the Player B receives the input. Now let us assume that at time 200 ms the Player B had reached a cover, run behind a wall. There are now three different perspectives with different outcomes. Figure 2.4 illustrates this case. From Player A's perspective the shot hits Player B. From Player B's perspective the shot does not hit because there is a wall blocking the shot.

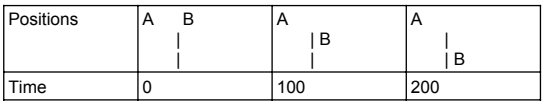

Figure 2.4: Three perspectives

The server's perspective depends on the Player A's position and direction and Player B's position in the state of the server at time 100 ms. This discrepancy needs to be addressed in the game's design by deciding which perspective will be the decisive one.

Lee and Chang [LC18] propose a scheme where the shooter's perspective is used but the target has a change to appeal the decision. This is to prevent a situation where a player has taken cover but gets shot anyway due to latency. The server first decides if it was a hit from the shooter's perspective. Then, if it was a hit, the server sends this information to the target client. The target client then responds either confirming or denying the hit. Then the server sends the final decision to all clients. This adds a delay to the process in the amount of the target client's round-trip time plus the intra-end-host processing times. When the server receives an input for a shot, it rolls the state back to the shooter's perspective, and when it receives an appeal, it rolls the state back to the appealing client's perspective. This is to make an authoritative decision and to prevent cheating.

As an essential aspect in delay compensation, Savery et al. [Sav+14] consider that it is important that the local view is kept intuitive and that the game critical decisions produce outcomes that can be rationalized by the players. This means that, for some parts in some points in time, it is acceptable that the states in different machines diverge. Important is how it appears to the players. For example, it may be acceptable that the same object is at slightly different position on different players' machines. This can occur as a result of using dead-reckoning to move the object smoothly. As a trade-off, consistency between players is reduced, but a smooth and intuitive movement of the object is achieved. It may be thus beneficial to sacrifice some game state consistency to achieve a more intuitive local view.

Ultimately, network delay can not be removed completely, due to speed of light setting a hard limit on the bit propagation speed, but the various in-game delay-compensation methods can reduce the visible effect of the network delay. The interaction between players may not be instant but delay compensation methods can make it seem as instant. In-game delay-compensation methods are thus an important element in the overall delay management of multiplayer games.

3 Network Support for Managing Delay

In this chapter we will take a look at Active Queue Management (AQM) and differentiated packet treatment as the main techniques in managing delay within the network. From the AQM algorithms we will consider Random Early Detection (RED) [FJ93], Proportional Integral Controller (PIE) [Pan+17], Controlled Delay (CoDel) [Nic+18] and FlowQueue CoDel (FQ-CoDel) [Høi+18]. RED is a well known AQM algorithm originally described in a paper from 1993. PIE, CoDel and FQ-CoDel are more recent ones that the Internet standards organization IETF has shown interest in. Regarding differentiated packet treatment we will take a look at packet scheduling in general and then in more detail the Differentiated Services (DiffServ) architecture.

3.1 Active Queue Management (AQM)

AQM is a technique to control queues in routers and switches. AQM algorithms attempt to provide sophisticated methods on detecting when to drop packets in order to keep the queues small. This is in contrast to passive techniques such as drop tail. Drop tail algorithm lets the queue fill completely and then drops arriving packets until there is again more space in the queue. AQM algorithms instead drop packets even though there is room in the queue in order to provide better performance such as lower latency.

RED

Random Early Detection (RED) is an AQM algorithm that drops packets probabilistically based on average queue length [FJ93, pp. 1–6]. RED measures the amount of packets, or optionally the amount of bytes in the buffer on average and decides upon three options: If the average queue size is small enough, that is, below a certain minimum threshold, no packets are dropped; If the average queue size is big enough, that is, above a certain maximum threshold, all incoming packets are dropped; If the average queue is between those thresholds a probabilistic dropping algorithm is applied. The closer the average queue is to the maximum threshold the more probable it is that a packet is dropped. Also, per each consecutive non-dropped packet the probability of the next packet to become

dropped is increased.

RED does not separate flows from each other and therefore it drops randomly a packet belonging to any flow [FJ93, pp. 1–6]. One host may send a lot of traffic and congest the buffer while another host may send only a small amount of traffic. Still, a packet from the smaller flow may end up being dropped during congestion.

However, the probability of a particular flow being selected is affected by the configuration of RED and the traffic characteristics of different flows [FJ93, pp. 5–6]. If RED is configured to measure queue by the packet amount and there is a flow that sends packets more than the other flows, then a packet from this flow is more probable to become dropped. If RED is configured to measure queue by the byte amount and there is a flow that sends more bytes, then a packet from this flow is more probable to become dropped.

PIE

Proportional Integral controller Enhanced (PIE) is an AQM algorithm that drops packets probabilistically based on average queue latency [Pan+17]. By focusing on the queue latency instead of the queue length PIE aims to bring better solutions to the bufferbloat problem.

The average queue latency is calculated by dividing the average queue size by the average departure rate [Pan+17]. This is an implementation of the Little's law [Bha15, p. 206]. It provides memory benefits since per packet timestamps are not required to calculate an average [Pan+17].

The probability for packet drop is set by gradually adding or subtracting from the previous probability value depending if the current average queue latency is above or below a target value [Pan+17]. If the queue latency is above the target the drop probability needs to go up in order to drop more packets and bring the queue latency down, and vice versa. The further away the latency is from the target the bigger a change is applied to the drop probability. This is the idea in the proportional integral controller.

The velocity at which the latency is changing is also taken into account in the calculation of the drop probability [Pan+17]. For example, if the queue latency is far above the target but it is trending down fast, then no big adjustments are necessary to the current drop probability: with this probability we are already heading where we want to. But if instead the latency is not already trending down then big adjustments are in place. The drop probability is updated at a certain interval. The default interval is 15 ms.

PIE is designed to well accommodate small bursts during a time of low link utilization [Pan+17]. If no congestion has been experienced for a while, packet dropping will be switched off completely for a certain period of time. By default, the period is 100 ms. This mode kicks in when drop probability has gone down to zero and the current and the previously measured average queue latencies are both less than half the target latency.

CoDel

CoDel is an algorithm that manages traffic by dropping packets based on minimum queue occupancy times [Nic+18]. This is to reduce persistent queues. A persistent queue, also known as a standing queue, exists when the buffer never empties but instead some queue always remains in the buffer contributing to overall latency without providing any benefits.

CoDel measures how long each packet stays in the queue [Nic+18]. This is called a packet sojourn time. If consecutive packet sojourn times are above a certain target for long enough (for an *interval*) the algorithm starts dropping packets. In other words, if during an interval one or more packet sojourn times is at the target or below, packets will not be dropped. Thus, there are four time related quantities to consider: an individual packet sojourn time, the interval, the target value and the time when a packet sojourn time last was below or same as the target. The interval and the target are constants. The recommended values are 100 ms for the interval and 5 ms for the target.

When the dropping state is entered, one packet is dropped and a time for a next packet drop is calculated [Nic+18]. The calculation is: interval divided by a square root of the amount of packet drops since the drop state started. This results in a dropping distance that shortens over time. In other words, the longer the packet sojourn times are over the target the more packets will become dropped per time unit. When a packet sojourn time is again at the target or below, the drop state is exited. This also resets the drop distance.

The tracking of the minimum packet sojourn times seeks to tackle the bufferbloat problem [Nic+18]. CoDel accommodates transient bursts but fights persistent, standing queues. It allows the buffer to be filled temporarily but starts dropping if the queue does not drain.

FQ-CoDel

FQ-CoDel [Høi+18] adds a flow queuing and a Deficit Round Robin (DRR) [SV95] scheme to CoDel. FQ-CoDel arranges packets to separate queues based on five-tuple of source IP address and port number and destination IP address and port number and the protocol

number stated in the IP header. The CoDel algorithm operates inside each queue and a modified DRR algorithm based on DRR++ [MS00] schedules packets from the queues.

Each queue is given a quantum of credits [Høi+18]. These credits represent the amount of bytes the flow can forward. The modified DRR scheduler decides from which queue packets are forwarded and upon the dequeue process the credits of the queue are subtracted accordingly. This flow queuing and DRR scheduling scheme enables fairness between flows. A flow sending a lot of traffic does not get to consume most of the link capacity but the capacity is shared evenly between all the flows, instead. The default quantum in FQ-CoDel is set to 1514 bytes.

In addition to providing flow fairness FQ-CoDel gives precedence to new flows, by having separate lists for new and old flows [Høi+18]. A flow is first put into the list of new flows. When the queue of the new flow runs out of credits or the queue becomes empty the flow is put into the list of old flows. If the queue becomes empty while in the head of the list of old flows, it will be removed from the list and is no longer considered active. If the queue is not empty, it is again given a quantum of credits and the flow is put back to the tail of the list of old flows.

The list of new flows is always emptied before the old flows are processed [Høi+18]. Thus, a flow generating traffic sparsely has a chance to be processed ahead of the bigger flows. The maximum sending rate to still receive precedence depends on the link bit-rate and the amount of other flows and their sending rate.

3.2 AQM and Multiplayer Games

In PIE and CoDel the size of a packet does not affect the probability on the packet becoming dropped [AC17]. A big packet is just as likely to become dropped as a small one. This may be unfair for game traffic which usually consists of frequent small packets. Some flow could be sending 1500 B packets per unit time while the game flow was sending 50 B packets but this difference would not be reflected in the dropping probability. Unlike PIE and CoDel, RED provides a dropping scheme that can be either packet or byte based [FJ93, pp. 5–6]. In this light RED might provide better fairness for game flows against competing traffic.

However, RED is sensitive to its parameters and finding a good setting is considered to be difficult [May+99]. For this reason, RED has experienced lack of deployment [KRW14,

p. 1]. Extensions to RED with automatic parameter tuning, such as self-configuring RED [Fen+99] or ARED [FGS01], aim to mitigate the parameter configuration problem.

Khademi et al. [KRW14] evaluate PIE, CoDel and ARED. They find ARED to provide performance nearly similar to PIE and CoDel, except only on light congestion ARED performed clearly worse. For CoDel, they find shortening the drop mode interval to be beneficial. In their test, shortening the interval from 100 ms to 30 ms and to 5 ms reduced the goodput* only slightly while the improvement in median queuing delay was significant (37.5%-54.8%). For PIE, the adjustment of the drop probability update interval from 100 ms to 30 ms or to 5 ms did not show as clear benefits: during light congestion, shortening the interval improved the queuing delay but, as an apparent trade-off, reduced the goodput; during moderate or heavy congestion, shortening the interval did not always improve the queuing delay and the goodput also changed only slightly. Considering latency requirements of multiplayer game traffic, shortening CoDel's interval from the default 100 ms would seem beneficial.

Regarding flow fairness issues, a multi-queue AQM scheme could provide a solution. FQ-CoDel divides different flows to different queues and uses DRR to provide fairness between them while also giving precedence to new flows. Armitage & Collom [AC17] test PIE, CoDel, FQ-CoDel and FQ-PIE with a traffic typical to a First-Person Shooter game competing against two queue building TCP flows. In the test FQ-PIE used a flow queuing scheme similar to that of FQ-CoDel's. They find that on the single-queue schemes the game traffic suffers from significant packet loss while on the multi-queue schemes the game traffic experiences almost zero packet loss. When the game traffic loss rate was 11-17% on a single-queue scheme, it was 0% on a multi-queue scheme.

Based on the experiments of Armitage & Collom it seems it would benefit networked multiplayer gaming if routers and switches implemented multi-queue AQM. However, there are some challenges that arise from combining the built-in queues of network devices to the queues of an AQM algorithm [Cab14, p. 10]. It may result in too many queues and too much complexity on the hardware. An AQM algorithm called CAKE [HTM18] which is designed as a refinement of FQ-CoDel tries to alleviate the complexity by utilizing set-associative hashing [HH07, p. 474], a method commonly used in caching. With set-associative hashing, CAKE can have fewer queues than FQ-CoDel but still achieve the same hash collision probability [Høi+18, p. 14]. No IETF RFC has yet been published

*Goodput is the amount of useful application level data delivered from the sender to the receiver per unit time

on CAKE, however. PIE, CoDel and FQ-CoDel are implemented in Linux kernel onwards from late 3.x series and a variant of PIE is used in devices following the DOCSIS 3.1 standard [AC17]. FQ-PIE will be implemented in Linux kernel 5.6 [Ram20].

3.3 Packet Scheduling

Network devices use scheduling to control which packets are forwarded from the buffer [Bri+16, pp. 2168–2169]. An example of a simple form of scheduling is first-in-first-out (FIFO) scheduling. In FIFO scheduling each packet gets the same treatment and the delay in forwarding affects each packet similarly. However, multiple more sophisticated methods exist. They can give differentiated treatment to packets and they can also bring fairness to the traffic with various criteria. Some concepts for scheduling are class or flow based, latency specific or hierarchical scheduling. The actual implementations can be combinations of multiple concepts.

In class based scheduling packets are treated differently based on what class they belong to [Bri+16, p. 2168]. This requires some method to classify the packets or the flows. Network devices then implement treatment aggregates where each class belongs to some treatment aggregate. A policy determines the buffer space and scheduling of packets within each treatment aggregate with the intent to provide a certain quality of service.

The class based scheduling can be implemented as a router/host-based model where the treatment is provided only in the context of a single router or host [Bri+16, p. 2168]. Alternatively, in Integrated Service (IntServ) model or in Differentiated Service (DiffServ) model the treatment is intended to be provided across a domain [Bri+16, p. 2168]. In the IntServ model, the RSVP protocol [Bra+97] is used to request the nodes along the path to reserve adequate capacity beforehand, in order to provide a requested quality of service for an incoming flow [BCS94]. In the DiffServ model, the network packets carry information of their expected service class using Differentiated Services Code Point (DSCP) in the IP header, and nodes on the path then separate packets to different treatment aggregates based on the DSCP [BBC06; Bak+98].

The IntServ model may be difficult to implement in the Internet because it requires reserving capacity for the whole path [Aur+20]. In the DiffServ model, instead, the service is provided by domains separately from each other with each having their own policies. This can make the DiffServ model more scalable and thus easier to implement in the Internet. However, the resulting quality of service may be more uncertain with DiffServ than with

IntServ [GR16].

Flow based scheduling separates traffic belonging to different flows [Bri+16, pp. 2168–2169]. The purpose is to reduce the effect that flows can have on each other. For example, Fair Queuing aims to provide a fair share of the link capacity for competing flows. Conceptually, a flow sending twice as much data per time unit than another flow would not get to use twice as much of the capacity of the link, but instead both would use roughly the same amount. This would benefit the smaller flows. Techniques such as Round Robin can be used to forward data fairly from multiple queues.

Latency specific scheduling methods try to provide low or defined latency [Bri+16, p. 2169]. Last-In-First-Out (LIFO) method forwards new packets while packets that have ended up in the queue have to wait. This minimizes delay for new packets but maximizes delay variance. Shortest Queue First (SQF) uses class or flow based scheduling and forwards packets from the queue that is the shortest. This benefits thin or short flows.

Hierarchical scheduling provides different hierarchy levels where some traffic is given preference over some other [Bri+16, p. 2169]. Packets can be separated into different hierarchy levels based on which flows or classes they belong to. Preferential treatment can be realised by using the Differentiated Services or the Integrated Services.

Differentiated Services

Differentiated Services (DiffServ or DS) is a concept where packets are given different treatment based on their service class [BBC06; DH17]. This differentiated treatment is realized at network nodes the packet traverse. For example, a router may forward some packets immediately and queue some others, depending on the service they have requested. The service class is denoted by DSCP (Differentiated Services Code Point). For the IPv4 header, the six most-significant bits of the TOS field form the DSCP, and for the IPv6 header, the six most-significant bits of the Traffic Class field form the DSCP [Bak+98].

The sending application can set the DSCP bits as it wishes and thus request a certain service level [BBC06]. However, the network nodes forwarding the packet may, or may not, comply with this request. The nodes can also change the bits as they wish. For example, an AS (Autonomous System) may have its own DS bit scheme and at the ingress node to this network it may change the bits to better suit its needs. Hence there are no guarantees that the service class request is honored. The service class is only a request, a wish, rather.

The services that a network node may provide are defined as Per-Hop-Behaviors (PHB) [BBC06]. Examples of them are Default Forwarding (DF), Assured Forwarding (AF) and Expedited Forwarding (EF). DF is best effort, normal service. AF and EF are enhanced versions of DF: they seek to provide better performance than the normal level, such as less delay, less packet drops or less jitter. AF is for elastic traffic that can react to congestion. EF is for inelastic traffic that can not react to congestion.

The guideline [BBC06] for differentiated services defines twelve service classes. Each service class maps to a certain PHB. Two of the service classes are for network control and the rest of them are for user traffic. Network control category consists of, for example, communication between routers. User traffic category consists of, for example, application client-server communications. Each service class represents a certain type of traffic and the requirements it has. It may be for multimedia streaming, multimedia conferencing, low-priority data or signaling traffic, to name a few.

3.4 Differentiated Services and Multiplayer Games

For real-time traffic in general there are three notable service classes: Telephony, Multimedia Conferencing and Real-Time Interactive [BBC06]. These three classes have the strictest delay expectations. The Telephony class is, as the name suggests, for voice communications, such as VoIP, and for data communication that uses the voice channel, such as fax and modem. The traffic is expected to be small fixed-size packets sent at regular intervals. An admission control may restrict the use of this traffic class. The Multimedia Conferencing class is meant for video calls with variable sized packets sent at regular intervals. The sender is expected to reduce the size of packets if congestion is experienced. The Real-Time Interactive class is for variable size and variable rate traffic that can not be adjusted based on congestion. This class is recommended for video calls without rate control, and also for interactive games.

The DSCP marking for Real-Time Interactive class is CS4 [BBC06]. The PHB recommended for CS4 is EF with “relaxed” performance parameters. Whereas for Telephony the DSCP marking is just the plain EF and thus the PHB is the normal EF. For Multimedia Conferencing the recommended DSCP markings are AF41, AF42 and AF43. They belong to the assured forwarding PHB group and denote different levels of service within it. The PHB is AF instead of EF since Multimedia Conferencing streams are expected to be able to adjust their sending rate.

A network (intermediary) node complying to CS4 service class is expected to provide low delay, low jitter and low packet loss for the traffic [BBC06]. This requires that the node allocates enough bandwidth so that packet drops would not happen. It is recommended that this traffic is allowed to bypass an AQM queue since it can not react to congestion anyway. The network nodes need to monitor that a host does not exploit this preferential treatment by sending CS4 traffic in excess.

If a sender uses CS4 class and tries to send in an excessively high rate, it is likely that the packets begin to get dropped. Interactive games, however, may need to send only a relatively small amount of data per each state update and for this reason they might be suitable for the CS4 class and get to bypass the AQMs and see minimum delays.

However, despite the EF PHB being recommended for interactive games, the AF PHB could also be useful. There are four main service classes within the AF and they range from AF1x to AF4x [BJ15, p. 10]. These denote different forwarding treatments with higher number meaning better treatment. There are three sublevels within each AF class, such as AF11, AF12 and AF13. They denote a drop precedence: lower number is favored over higher, that is, AF13 is dropped before AF12 [Wei+99, p. 2]. The AF4x class is the only one of the AF classes meant for real-time interactive traffic [BBC06]. The others have more relaxed latency constraints. The AF41, AF42 and AF43 classes could therefore be suitable for multiplayer game traffic.

Carrig et al. [CDM06] propose a scheme where the different AF service classes could be used for balancing latencies between players. A player who has a higher relative latency would request better AF treatment and a player who has a lower relative latency would request worse AF treatment. Similar solution is proposed by Liang et al. [LZC05] with some differences in the internal communication scheme. A study by Howard et al. [How+14] shows that the quality of experience of well connected players is hampered by poorly connected players. They recommend that improving the connection of the most lagged player should be the priority. The AF class might provide a suitable structure for fine tuning the latencies.

Ultimately, it can not be trusted that the network provides the requested service level. Intermediaries may bleach or modify the DSCP bits. In some cases, requesting a certain service level may even result in packets getting dropped; the operator not supporting the requested service level may decide to drop all such packets [CSF18, p. 87]. Therefore, asking for better may lead to getting worse. In this sense, for some cases, using the default service may turn out to be the best solution.

3.5 Impact of ECN

Explicit Congestion Notification (ECN) [FRB01] is a protocol extension for providing information of congestion without requiring a packet drop. Commonly, intermediaries such as routers and switches manage queues by dropping packets during congestion, the senders then infer the congestion from packet loss and react by reducing their transmission rate. ECN provides a mechanism for the intermediaries to explicitly notify the sender that there is congestion on the path. This removes the need to use packet drop as a means of informing senders about congestion.

ECN communication happens both on the IP layer and on a higher layer [FRB01]. An intermediary marks a flag in the packet IP header for incipient congestion. The receiver (end-host) echoes the information back to the sender (end-host) and the sender notifies the receiver that it has reacted to the information. The communication between end-hosts happens at a higher layer such as at the transport layer.

The ECN codepoint at the IP header is the two least-significant bits in the TOS field of IPv4 or Traffic Class field of IPv6 [FRB01]. The codepoint '00' (Non-ECT) denotes non-ECN-capable transport. The codepoint '10' (ECT(0)) or '01 (ECT(1)) denote ECN-capable transport. The codepoint '11' (CE) denote encountered congestion. The sender sets the codepoint ECT(0) or ECT(1) when ECN is used for the transport and an intermediary modifies the bits to codepoint CE when experiencing congestion.

If the endpoints use TCP as a transport protocol the receiver echoes the IP header CE flag to the sender by setting an ECN-Echo (ECE) flag in the TCP header of an acknowledgement (ACK) message [FRB01]. The sender learns from the ECE flags in the ACK messages that there is congestion on the path and reduces its transmission rate. The receiver keeps on setting the ECE flag until it has received a notification from the sender that it has reduced its transmission rate. The sender notifies the receiver that it has reduced its transmission rate by setting the Congestion Window Reduced (CWR) flag in the TCP header.

In order for ECN to work properly it requires support from the end-hosts and from the intermediaries on the path [FRB01]. The sender needs to be able to set the ECT(0) or ECT(1) codepoint. The receiver needs to be able to read the ECN bits in order to see if the CE flag is set or not. The receiver needs to be able to communicate to the sender if the CE flag was set, and the sender needs to be able to communicate to the receiver that it has reduced its transmission rate. An intermediary on the path needs to be able to set

the CE flag when experiencing congestion and all the intermediaries on the path need to not inadvertently modify or bleach the ECN bits.

With properly functioning ECN, packet losses and associated delays may be reduced [FRB01; LJS17]. The intermediary experiencing incipient congestion does not need to start dropping packets but instead it can CE flag them. This saves the transmissions from packet loss recovery delays or loss of data. When using a reliable transport, a lost packet (or the information in that packet) would need to be retransmitted, which would add the time it takes to detect the loss and the time it takes to retransmit the packet to the overall end-to-end delay. Also, HOL blocking delays might be experienced due to packet reordering. When using a non-reliable transport, a lost packet would not need to be retransmitted but the information in the packet would be lost (unless some redundancy is used in the transport). In multiplayer games, packet loss can create gaps into the sequence of update messages and have an effect similar to network delay [BF10]. With ECN, some of the packet losses and their effects can be removed.

If ECN were to be used with UDP the communication between end-hosts would need to be implemented at an upper layer [EFS17]. Unlike TCP, UDP does not provide built-in support for ECN. This is understandable since UDP is designed as a connectionless protocol where no feedback is provided from the receiver to the sender [Pos80]. With UDP, the communication where the receiver notifies the sender of the congestion and the sender notifies the receiver that it has reduced its transmission rate, would have to happen at the application layer or at some other layer above UDP.

In order to use ECN both end-hosts need to support it within their operating systems [FRB01; EFS17]. In case of using TCP, it is not necessary to expose the ECN bits to the higher layer since TCP and IP can take care of the communication by themselves. With UDP, the higher layer (at the receiver), such as the application layer, needs to be able to read the IP layer ECN bits in order to be notified of possible congestion. Also, the higher layer (at the sender) needs to be able to set the IP layer ECT(0) or ECT(1) bits to enable the use of ECN in the first place, unless it is enabled by default by the operating system.

The use of ECN requires, in practice, an AQM algorithm [BF15]. With AQM, the queue is not allowed to become completely full before the algorithm starts to react to the impending congestion. This makes it possible to ECN-CE flag and keep a packet instead of dropping it since there is still room in the queue. With a conventional queue management algorithm, instead, the algorithm would react only when the queue already overflows. An example of a conventional queue management algorithm is the tail drop (or drop tail) algorithm

which drops incoming packets when the queue is full. For the packet selected to be dropped ECN would be of no use since, at queue overflow, there is no room, the packet has to be dropped.

Among the AQM algorithms, RED, PIE and CoDel can be configured to use ECN [FJ93; Pan+17; Nic+18]. In FQ-CoDel, ECN is enabled by default [Høi+18]. FQ-CoDel also offers an option to set the threshold for ECN-CE marking lower than what the threshold for packet drop is, in order to react earlier to congestion.

When both ECN-capable and non-ECN-capable flows share a heavily congested link the situation may be considered unfair for non-ECN-capable flows. Armitage & Collom [AC17] observe that in a situation where all packets need to be marked for congestion the result is logically that the ECN-capable packets become marked with the ECN-CE bit and all others become dropped. For example, in a case where game packets use UDP and the end-host(s) do not support ECN on UDP traffic, all the game packets would become dropped in the aforementioned situation. At the same time, the competing ECN-capable flows would dominate on the link. In case the game traffic end-hosts and the network nodes on the path support ECN on UDP traffic, then, naturally, the packets may avoid becoming dropped during congestion.

Since ECN is foremostly designed to be used with TCP, support for ECN on TCP transmissions can be expected to be implemented within the network stack in operating systems. However, for UDP transmission, the accessibility of ECN bits is more uncertain. Based on documentations [Mic19a; Mic19b; QUI19] from Microsoft and from the IETF QUIC Working Group, the current Linux and Apple's operating systems allow application to read and write the ECN bits, whereas Windows operating systems allow application only to read ECN bits but not write them. Armitage & Collom [AC17] comment in general that ECN has not yet seen significant deployment in the Internet. However, the deployment is gradually increasing [Che17].

4 Protocols for Web Browser

In the following we describe WebSocket, QUIC, HTTP/3 and WebRTC. We consider their features and the congestion control mechanics they implement. At the end of this chapter we discuss how the protocols could be applicable as game data transports.

4.1 WebSocket

WebSocket is developed by HyBi Working Group and it is defined in RFC 6455 [MF11]. It is an application level protocol for bidirectional communications between client and server. Bidirectional means in this context that both the client and the server can send messages to each other over the same connection and the messages can be sent without preceding explicit request from the other (unlike, for instance, in HTTP/1.1). WebSocket traffic is carried on top of TCP.

In the case that the client and the server are using HTTP/1.1, WebSocket connection is established in the following way [MF11, pp. 6–9, 14–27]. WebSocket connection is initiated atop TCP connection with an HTTP upgrade request to a server. WebSocket client sends the message to a well known port on the server (port 80 or 443). The message is a request to upgrade the HTTP connection to WebSocket connection. It includes a WebSocket protocol version number and a challenge key. The challenge key is to confirm that the server actually understands this version of the WebSocket protocol. The server replies with a HTTP 101 Switching Protocols message which includes a valid answer key and the WebSocket connection gets established on top of the TCP connection. If the upgrade request is initiated by a script whose origin is different from the host receiving the request then Cross-Origin Resource Sharing (CORS) rules come into play and the host needs to opt-in for CORS or deny the request [MDN20b; Gri13, pp. 299–300]. If the host, that is, the server, accepts the request the response also needs to include a CORS header. After the WebSocket connection is established no HTTP traffic is required between the endpoints.

For HTTP/2 the establishment of a WebSocket connection is a bit different from that of HTTP/1.1's [McM18]. HTTP/2 uses stream multiplexing and for that reason the WebSocket connection becomes one of the multiple streams under a single HTTP/2 connection.

The HTTP connection is not upgraded but remains the same. One of the streams only becomes a WebSocket stream and other streams can remain what they are such as HTTP streams or other WebSocket streams. For this reason a HTTP upgrade request is not sent but a CONNECT request including a :protocol pseudo header with a value "websocket" is sent instead by the client. If the server supports this extension of the HTTP/2 protocol it can proceed with the connection handshake. Challenge key that was required with HTTP/1.1 is not used here. Instead the :protocol pseudo header replaces its function. The WebSocket protocol number used with HTTP/1.1 is also used with HTTP/2. After the connection is established, the WebSocket protocol operates inside the HTTP/2 stream in the same manner as if it was set up directly atop TCP by HTTP/1.1.

Some failure situations with WebSocket can occur when the server or intermediaries do not understand the protocol [Gri13, p. 301]. The concept of the challenge key is to handle this situation with the server. However, intermediaries could still misinterpret WebSocket frames and buffer or modify them unintentionally or consider them as faulty HTTP. To prevent this, Secure WebSocket (WSS) could be used and thus the intermediaries would not get to read the frames.

Compared to HTTP protocol, which in principle requires a client request in order for the server to send a message (except with unsolicited server push feature), WebSocket offers reduced delay for bi-directional communication [FR14, p. 6],[BPT15, p. 5],[MF11, p. 5]. Both parties can send a message right away, when desired, without any wait. This is, provided that TCP congestion window and receiver window allow sending at that time [BPA09, p. 3]. In this sense WebSocket can be thought of as a real-time communication protocol.

Congestion Control

Transport protocols use congestion control to regulate the amount of data they send to the network. Congestion control is necessary in providing a fair share of the bandwidth to different users and preventing adverse conditions and situations such as congestion collapse in the worst case.

WebSocket uses TCP which means that WebSocket traffic is controlled by TCP's congestion control algorithm [MF11; BPA09]. TCP is a transport layer protocol that provides reliable and in-order delivery [Pos81]. The algorithm deduces network congestion from packet loss, and packet loss is deduced from duplicate acknowledgments and timeouts.

There are numerous extensions defined for TCP congestion control and they are widely adopted. Here, however, we describe the basic version from its essential parts.

TCP congestion control consists of four main algorithms: slow start, congestion avoidance, fast retransmit and fast recovery [BPA09]. Slow start is used when the network conditions are unknown such as at the beginning of a connection or after a long idle period. It is also used after an acknowledgment from the receiver has not come in a certain time and a retransmission time-out occurs. In slow start, TCP increases the sending rate (roughly) exponentially [Ste97, p. 2]. The amount of data that is allowed to be sent is controlled by congestion window [BPA09].

At some point, slow start threshold is reached and TCP moves from slow start to congestion avoidance [BPA09]. This threshold is set dynamically based on previously experienced packet losses. From here on the sending rate is increased (roughly) linearly, the increase being approximately one packet more per round-trip time. The idea is to increase the rate more gently now because the threshold indicates that the maximum capacity of the path may be near.

When a packet reaches the receiver, the receiver sends an acknowledgment back to the sender, thus acknowledging the received segment [BPA09]. The original packet or the acknowledgment may become dropped in the network and the retransmission time-out may be reached. In this case TCP goes back to slow start and the aforementioned threshold is adjusted to reflect the current network conditions.

Fast retransmit and fast recovery aim to prevent the need to wait for the retransmission time-out [BPA09]. This is done utilizing duplicate acknowledgments. If there is a gap in the received data the receiver keeps on acknowledging the segment preceding the gap even though it receives new data. The arrival of three duplicate acknowledgments denotes a lost segment and the segment gets retransmitted right away. After the segment is acknowledged the threshold is adjusted as in the retransmission time-out case and TCP continues in congestion avoidance.

The reasoning for fast retransmit and why TCP does not enter slow start is that the acknowledgments are being generated which means that packets are going through and that means that it is likely that the network is not immensely congested [BPA09, p. 9]. Therefore TCP can continue sending at a higher rate than what it would if it went to slow start.

One notable extension to take into account is TCP Selective Acknowledgment (SACK)

[Flo+96]. By using TCP SACK the receiver will get a better understanding of what segments the receiver has actually received. The basic TCP only tells the last segment before the gap but the sender does not know what is received after that. TCP SACK provides the sender extra information where it can see what is received and where the gaps are. This way the sender can more efficiently send only the actual missing ones.

4.2 QUIC and HTTP/3

QUIC is a bidirectional byte-oriented protocol that runs on top of UDP [IT20]. QUIC is a transport protocol specification that defines streams, packets and connections but it also is an application layer protocol in the sense that its place is above a transport layer protocol, UDP, in the protocol stack. It is implemented in the user space, meaning that it does not require changes in operating systems and middleboxes. QUIC is therefore half a transport layer protocol and half an application layer protocol. Alongside with QUIC comes HTTP/3. HTTP/3 is the actual application layer protocol and it is designed to use QUIC as a transport.

HTTP/3 is meant to be a successor for HTTP/2 [Bis20]. The two have many similarities such as server push and stream multiplexing. One major improvement over HTTP/2 is, however, that HTTP/3 streams do not block each other: An individual stream may experience head-of-line blocking but this will not affect the other streams. In HTTP/2 all the streams would get blocked. What makes this improvement possible is the use of QUIC and UDP. UDP does not provide in-order and reliability guarantees and QUIC makes the use of this. We will take a look at this and other QUIC's features and then take a closer look at HTTP/3.

QUIC streams

QUIC provides bidirectional stream multiplexing [IT20]. Both the client and the server can open new streams. A stream is either unidirectional or bidirectional. The maximum number of streams for a connection is 2ˆ62. The stream identifier numbers start from zero so the biggest stream number is 2ˆ62-1.

All streams have in-order requirements [IT20]. The data is transmitted inside frames and if there are gaps in the sequence of frames that an endpoint has received it will block the delivery of the data to the application until the missing frames arrive. As mentioned

above, this only affects the stream in question and the other streams can continue. In HTTP/2 this is not possible since the transport protocol is TCP and it will block all the packets belonging to that connection. UDP does not have rules for delivery order so QUIC can define its own rules on top of it. In the operating system's protocol stack the transport layer (UDP) then lets all the packets through to the upper layer regardless of their order. At that upper layer is QUIC then checking the order. This way the custom rules can be enforced at the user space and no changes are needed in the operating system or in the network devices. QUIC comes shipped within the network browser so only an update to the browser is required at the client machines.

Streams are opened by sending a message and giving the message a stream number that has not been used previously in that connection [IT20]. A single message can open, carry payload and close a stream if desired. Thus, opening a stream does not entail any handshake operation; the stream opening is done implicitly. The receiving peer assigns the sender a certain maximum number of streams that the sender is allowed to open. If the sender runs out of available stream numbers it can request more from the receiver.

The two least significant bits of the stream number indicate if the stream is unidirectional or bidirectional and if it was initiated by the client or by the server [IT20]: If the least significant bit is zero, the stream is client initiated. If the bit is one, the stream is server initiated. If the second least significant bit is zero, the stream is bidirectional. If the bit is one, the stream is unidirectional. This means that the first client initiated bidirectional stream has a stream number of 0 and the next has a stream number of 4 and the next one has 8 and so on.

QUIC connections

QUIC provides 1 RTT and 0 RTT connection establishments [IT20]. In 1 RTT the endpoints declare connection parameters such as supported protocol version or maximum number of streams. A shared secret is established using Transport Layer Security (TLS). With 0 RTT the client can already send application data along with the first handshake message. 0 RTT provides less security guarantees than 1 RTT but it reduces the delay.

QUIC uses connection IDs [IT20]. Each connection has its own ID, or multiple IDs, and if the IP addresses and port numbers change during the connection, for example due to NAT rebinding, the connection ID helps in associating the new address or port to the old connection. An endpoint issues new connection IDs to its peer. This is done in the initial

message of the connection, and after that more IDs can also be sent. During the lifetime of the connection, IDs can be retired and new ones can be picked. If a peer runs out of IDs it can request more from the other peer. Changing IDs reduces the chance of linkability of the connections for an outside observer [DB20, p. 3].

When initiating the connection the client indicates the version of QUIC it wishes to use [IT20]. If the server does not support this version it replies to the client with a list of versions it supports. Client then chooses from these. If there is a version in the list that the client supports it initiates a new connection using that version.

QUIC packets

QUIC carries application data in frames [IT20]. Frames are contained in packets. Multiple frames can be coalesced into a single QUIC packet and multiple packets can be coalesced into a single UDP datagram. There are multiple frame and packet types.

The packet types can be divided into two main groups: long header packets and short header packets [IT20]. Long header packets are used for connection establishment and short header packets are used after the connection is established. Long header packets include the 0 RTT packet which can be used for sending application data. Otherwise application data is sent in short header packets. The type of the frame that holds application data is a STREAM frame.

QUIC is a byte-oriented protocol which means that it receives a byte stream from the sending application and arranges the bytes inside the frames and at the receiver side delivers the bytes to the application in the same order [IT20]. QUIC only sees the bytes coming in from the application and therefore does not have a notion of a complete application message. Thus it is a byte-oriented protocol (and not a message-oriented protocol).

HTTP/3

When a client is initiating a HTTP/1 or HTTP/2 connection the server can use HTTP Alternative Services (Alt-Svc) to advertise that HTTP/3 protocol is supported [Bis20]. If the client supports it too the HTTP/3 connection can be established. The actual underlying connection is a QUIC connection and then on top of it the endpoints send HTTP/3 SETTINGS frames to each other to start the HTTP/3 session.

The basic interaction in HTTP/3 (and in HTTP/2 and HTTP/1, too) is the client sending

a request and the server sending a response [Bis20]. For example, a client requests a HTML file and the server responds with the HTML file. In HTTP/3 (and also in HTTP/2) the server can push the response to the client without the client explicitly requesting it. This can be done when the server anticipates that the client will soon make that particular request anyway. The benefit here is reduced delay. By the time the client discovers that it needs to make the request, the data is already available at the client. For example, the client requests a HTML file, the server responds with the HTML file and the server knows that the HTML file references some image files, so it pushes the image files to the client even though the client did not yet request them.

HTTP/3 uses QUIC streams to multiplex the requests [Bis20]. Each request opens its own bidirectional stream where that particular transaction is performed. The request and the associated response travel on the same stream: they have the same QUIC stream ID. The type of this stream is a request stream, although it also carries the response. Since QUIC streams are multiplexed and do not block each other, multiple requests can be carried out concurrently.

Server push opens its own stream [Bis20]. This stream is a server initiated unidirectional stream, called a push stream. The request that caused the server to do the push runs on its own separate stream, on the request stream. The server sends a PUSH_PROMISE frame to the request stream. The PUSH_PROMISE consists of a Push ID and a request header. The Push ID is a reference to the push stream. The request header is the same as the request header that the client would use to request the content that was now pushed. When the client is about to make a request it finds that the request matches with the request header carried inside the PUSH_PROMISE frame. It reads the Push ID from the PUSH_PROMISE and matches it with the push stream that has the same Push ID. The pushed content in that stream is the response to the request that the client would have made.

In addition to request and push streams HTTP/3 has control streams [Bis20]. Each of the endpoints open a control stream at the start of the connection in order to send the SETTINGS frame. During the connection lifetime the control stream can be used for indicating how many pushed responses the client allows the server to send or if it wants it to cancel a particular push. The server uses the control stream to initiate a graceful shutdown of the connection by telling the client to not send more requests.

The requests and responses (including push responses) consist of HEADER frames and of possible DATA frames [Bis20]. The header is mandatory. Then come optional data

frames. Lastly comes optional trailing headers. HTTP/3 endpoint does not have to receive a complete message before it can start reading it. The server can start sending a response before it has read the whole request if it decides it has received enough information to form the response. A single request and a response, or a server push, consumes the stream. For the next request or push, a new stream is opened.

QUIC offers four types of streams: client or server initiated unidirectional or bidirectional streams [Bis20; IT20]. However, HTTP/3 does not use the server initiated bidirectional streams. The server only opens a control stream and the push streams, and these are unidirectional. Also, HTTP/3 does not use out-of-order delivery. QUIC is an in-order transport protocol. However, the design documents (QUIC draft 27, chapter 2.2.; HTTP/3 draft 27, chapter 6.) mention a possibility for out-of-order implementation. Nevertheless, HTTP/3 implements only in-order delivery of data within streams. Between streams, however, the data will be delivered unordered.

All HTTP/3 frames have a type, length and a payload field [Bis20]. The payload depends on the type of the frame. HTTP/3 has a total of seven different frame types (SETTINGS, HEADERS, DATA, PUSH_PROMISE, MAX_PUSH_ID, CANCEL_PUSH and GOAWAY, plus a reserved type range for undefined). A single HTTP/3 frame can span multiple QUIC packets.

Congestion Control

QUIC implements flow control and congestion control to regulate the rate and amount of data to be sent. Flow control is meant for preventing the sender from sending more data than the receiver can handle [IT20, pp. 24–29]. Flow control consists of the receiver sending messages to the sender where it tells how much the sender is allowed to send. In this fashion the receiver regulates the maximum amount of streams the sender can open, the maximum amount of data the sender can send in a particular stream and the maximum amount of data the sender can send in all streams combined. The receiver periodically sends these allowance credits and tries to balance between sending a small amount of credit often or sending a bigger amount of credit less often. Sending more often means more traffic and more overhead. Sending less often with more credit requires more resources to be allocated in anticipation of larger amount of data possibly to be received from the sender. If the sender runs out of credit it can request more from the receiver. This, however, would be an undesirable situation since it halts the transmissions until more credit is received, which will be at least a full round-trip time.

QUIC frames can be divided into two categories: ack-eliciting and non-ack-eliciting frames [IT20, pp. 88–89]. If a packet contains one or more ack-eliciting frames it will be an ack-eliciting packet. All packets are acknowledged, but the acks of non-ack-eliciting packets are only sent if there are acks of ack-eliciting packets to be sent: The non-ack-eliciting packets are acknowledged along with the ack-eliciting packets.

The packets are acknowledged by packet number ranges [IT20, pp. 91–92, 128–129]. The ACK frame contains the highest packet number to be acknowledged, followed by a range of preceding packet numbers also to be acknowledged. This is followed by a possible gap range denoting missing packets and that is followed again by an ack range denoting packets to be acknowledged, and so on. By alternating gap and ack ranges, further noncontinuous packet number sequences can be acknowledged.

The ACK frame also contains the Ack Delay (or Host Delay) and a count of ECN codepoints [IT20, pp. 92–93, 96–99, 127, 129–130]. The Ack Delay is the time the receiver intentionally took (processing etc.) between receiving a packet and sending the acknowledgment. This measurement is taken from the packet with the highest packet number. The count of ECN codepoints denotes the sums of ECT(0), ECT(1) and CE markings in the received packets.

When packets are lost QUIC does not resend the same packet [IS20, p. 6], [IT20, pp. 93–96, 133–134]. Instead, it resends the information carried in the frames of the lost packet, if it is still necessary. Thus, the new packet will not be the same as the lost packet. Also, QUIC uses strictly increasing packet numbers: the new packet will have a new packet number; packet numbers are never reused. Application data is ordered at the receiver based on the byte offset field of the STREAM frame and therefore the packet number does not need to reflect the original ordering.

QUICs congestion control is based on the basic TCP congestion control, TCP SACK and TCP NewReno, among other many variants of TCP [IS20, pp. 12–27]. It has the slow start, the congestion avoidance and the recovery period phase. It also has a notion of persistent congestion.

In slow start, congestion window (cwnd) is increased by the amount of bytes acknowledged [IS20, pp. 21–22]. After slow start, congestion avoidance is entered. In congestion avoidance cwnd is increased by a maximum packet size after a cwnd amount of bytes is acknowledged. If a packet is lost or the count of ECN-CE bits increases, the cwnd is set to half and slow start threshold is set to cwnd, and recovery period is entered.

In the recovery period, the lost data is retransmitted (alternatively, updated information may be sent, or retransmission may be abandoned) and also new data may be sent [IS20, pp. 21–22]. The cwnd is kept unchanged. When the sender receives an acknowledgment of any of the packets that were sent after the recovery period was entered, the recovery period becomes completed and QUIC continues with congestion avoidance.

QUIC congestion control considers a packet lost when it is behind a newly acknowledged packet in its packet number for a certain sufficient amount. A packet is also considered lost if it was sent a certain sufficient time before a newly acknowledged packet [IS20, pp. 12–14]. These thresholds are implementation specific, but recommended values are three packets or roughly two round-trip times behind.

Persistent congestion is experienced when a certain threshold is reached between the time an oldest and newest unacknowledged packet was sent [IS20, pp. 23–24]. This requires that a packet is acknowledged after them and no packet is acknowledged between them. The threshold is implementation specific. If a persistent congestion is experienced cwnd is set to minimum and slow start is entered.

Probe time-out is used for testing that packets go through and acknowledgments can be received [IS20, pp. 14–18]. Probe time-out happens when an acknowledgment for a packet latest sent is not received in a certain time. In this case one or two datagrams are sent in order to get a reply from the peer. This mechanism is useful especially in reacting to a tail loss situation where there are no packets being sent anymore after the lost one and therefore no acknowledgments to receive to reveal the loss. Probe time-out is similar to TCP retransmission time-out.

4.3 WebRTC

Web Real-Time Communication (WebRTC) is a technology that combines multiple standards and protocols to achieve peer-to-peer communications for browsers [Gri13, p. 309]. The main focus is on video and audio streaming, such as video calls, but it also provides a channel for delivery of arbitrary data. The underlying transport layer protocol is UDP.

The browser API for WebRTC was first published in 2011 [Alv11]. Now, the latest version of the API definition is from 22 September 2020 [JBB20]. The API is being developed by "Web Real-Time Communications (WEBRTC)" W3C Working Group [W3C20], whereas the underlying protocols are defined by "Real-Time Communication in WEB-browsers

(RTCWEB)" IETF Working Group [W3C20; Goo19] [Gri13, p. 310]. The IETF working group was concluded August 14, 2019 [IET20a]. The API definition is still on the W3C recommendation track with a status "Candidate Recommendation" which means it is not completed yet [ER20].

There are many JavaScript APIs, standards and protocols included in WebRTC [Gri13, pp. 309, 317]. The three main JavaScript APIs are MediaStream API for acquiring and outputting video and audio, RTCDataChannel API for transmitting arbitrary application data and RTCPeerConnection API for handling the connections. The main protocols and standards WebRTC utilizes are SRTP [Nor+04], SCTP [Ste07], DTLS [RM12], SDP [PHJ06], ICE [KHR18], STUN [Mat+08] and TURN [MRM10].

The Secure Real-time Transport Protocol (SRTP) is a variant of the Real-time Transport Protocol (RTP) with a security aspect [Nor+04, p. 3]. SRTP is used for delivering video and audio data [Sch+03, pp. 4, 19]. Alongside SRTP there is the SRTP Control Protocol (SRTCP). It provides statistics of the data delivery so that the sender side can make appropriate adjustments such as adjust video encoding based on interpreted congestion.

The Stream Control Transmission Protocol (SCTP) is used in WebRTC for the Data Channel which delivers the arbitrary application data [Gri13, p. 344]. SCTP allows in-order and out-of-order delivery of data [Ste07, p. 89]. The basic version provides a reliable transport. With an extension the protocol can also be made partially reliable or unreliable [Ram+04; Tüx+15].

Datagram Transport Layer Security (DTLS) provides equivalent security as the widely adopted Transport Layer Security (TLS) protocol but it is designed for datagram traffic [RM12]. TLS requires reliable delivery of packets whereas DTLS can work in an unreliable channel. Thus, it is the choice for SCTP transmissions and also for the handshake procedure of a SRTP connection in WebRTC [Gri13, pp. 343–344], [MR10, p. 3], [Res19, p. 11].

Session Description Protocol (SDP) provides a standard way to describe metadata about a session [PHJ06, p. 3]. In WebRTC SDP is used in a negotiation of a peer-to-peer connection where it describes properties of the data the peer is about to send and other essential information regarding the connection [Gri13, p. 323].

Interactive Connectivity Establishment (ICE) is a protocol for establishing peer-to-peer connections behind NATs [KHR18]. This is done with the help of a protocol called Session Traversal Utilities for NAT (STUN) which enables the discovery of endpoint's public IP

address and port number [Mat+08]. By using STUN servers ICE tries to find address and port tuples that could enable a connection between the peers [KHR18, p. 6]. After finding candidates it tests to see if connection can be made. If all the attempts to create peer-to-peer connection fail a TURN server can be used [MRM10]. Traversal Using Relays around NAT (TURN) is a protocol for connection that uses a relay server (TURN server) between the peers. The peers may not be able to connect to each other directly because of NATs but they can connect to a server in between and the server can relay their traffic.

Connection establishment

Peer-to-peer connection establishment starts by the endpoints first expressing to each other that they wish to start a connection [Gri13, pp. 317–334]. In web client-server communications this would happen by the client making a DNS lookup and sending a message to the address and to a well known port of the server. In a peer-to-peer context this does not work since the peer does not know the other's address and port. A signalling channel is required. The peers communicate through this signalling channel to negotiate the parameters of the connection.

WebRTC standard does not specify any particular signalling solution and this is left for the application to decide [Gri13, pp. 317–334]. What WebRTC does is it exposes the RTCPeerConnection API to which the application registers the local streams (audio, video, data) it wants to use in the session. When the streams are registered RTCPeerConnection creates an SDP offer description about the streams. The offer may consist of information about the bit-rate and codecs used for the video and audio, among other things. Then it sends it via the signaling service to the other peer. The other peer then creates an answer consisting of its own streams and it is delivered back to the originating peer. Now both parties have information about the session and they can proceed establishing the connection.

RTCPeerConnection starts connectivity checks already after the SDP offer is created. The connectivity checks are done using ICE and, if required and configured, with STUN and TURN [Gri13, pp. 317–334], [KHR18, pp. 6–13]. The ICE tries to find possible public address and port tuples of the local peer, called candidates. When these are discovered they can be sent in an SDP offer for the other peer. They can be sent all at once with the first offer or they can be sent one by one when discovered. When receiving the remote peer's answer which contains the remote peer's candidates ICE has all the information needed for trying to find a working connection between the two peers. The connectivity

checks are done with the help of STUN. Once the connectivity checks are completed ICE's work is done and the peers can start sending data, provided that a connection was found. If not, TURN can be used.

During the lifetime of the connection the ICE process can be restarted [Gri13, pp. 331, 334], [KHR18, p. 13]. RTCPeerConnection may, for instance, periodically restart the ICE process in order to find improved connections. Also if streams are added or removed from the connection, by opening a new data channel, for example, an offer will be sent and a new ICE process will be started.

Data delivery

Video and audio data is delivered with SRTP protocol and arbitrary application data with SCTP [Gri13, pp. 337–344]. SRTP runs directly on top of UDP and DTLS is utilized only at the connection handshake. DTLS provides a shared secret that the endpoints can use as a keying material during the SRTP connection. For SCTP the DTLS is used for the duration of the whole connection. SCTP traffic is tunneled over DTLS, that is, all the SCTP packets are encapsulated inside DTLS packets [Gri13, p. 344], [Tüx+17, p. 3].

The local video and audio streams are acquired via the MediaStream API [Gri13, pp. 312–313, 333–334]. The user specifies the constraints, such as video resolution, and the MediaStream interface provides a stream which can then be given to RTCPeerConnection API and eventually be sent to another peer. When the remote peer's video stream is received it can be outputted through the MediaStream interface to the user's screen. Put another way: input can be local device or remote peer; output can be, for example, local video element or remote peer.

The transmission of video and audio is done over SRTP. The SRTCP provides feedback on the connection. The WebRTC media engine in the browser adjusts the quality of the video and audio based on the feedback. The engine also deals with packet loss, network jitter and congestion control. There are several congestion control algorithms proposed for RTP traffic (RFC: 8298, 8382, 8593, 8699, 8698) [IET20b; TES14; rje16]. The IETF Working Group RMCAT works to develop them and the work is ongoing. Detailed assessment of the audio and video mechanics of WebRTC is out of scope of this thesis and are not discussed further.

To send application data the Data Channel needs to be opened. Data Channel is established on top of a SCTP association (synonym for connection in SCTP's vocabulary) that

is established on top of a DTLS connection. First the DTLS handshake is completed and then the SCTP handshake [Tüx+17, p. 5]. The DTLS handshake takes two roundtrips, the SCTP handshake also takes two roundtrips, and the Data Channel handshake takes one roundtrip [Gri13, p. 337], [JLT15a, pp. 3–4], [Ste07, pp. 56–57]. On SCTP, application data can already be sent during the second roundtrip of the handshake, and on Data Channel it can be sent right after the initial handshake message without waiting for a reply. The Data Channel handshake and data messages can be bundled into the same SCTP packet and therefore it takes a minimum of total three roundtrips before application data can be sent.

The properties of the Data Channel are set in the handshake message [JLT15a, pp. 5–6]. Data Channel can have reliable or unreliable transmission, it can have in-order or out-of-order delivery and its priority can be set. The priority scheme includes four levels to choose from: below normal, normal, high and extra high [JLT15b, p. 10].

Data Channel is encrypted, message-oriented and bidirectional [JLT15b, pp. 10–11], [Ste07, p. 91]. SCTP packets carry complete messages and fragmentation is only done if required due to message oversize. The data can be binary or UTF-8 and both parties can send messages at will. The overhead of Data Channel's protocol stack (IP, UDP, DTLS, SCTP) is around 100 bytes [Gri13, pp. 351, 355]. WebRTC allows opening the Data Channel with audio and video streams disabled.

Data Channel Congestion Control

Data Channel uses SCTP and therefore it uses SCTP's congestion control. SCTP's congestion control is based on TCP's congestion control defined in RFC 2581 (that is since obsoleted by RFC 5681) [PAS99, p. 1], [Ste07, pp. 94–100]. SCTP utilizes selective acknowledgments to report precisely where the gaps in the received data are, which is similar to TCP SACK. SCTP has the same four modes as TCP has: slow start, congestion avoidance, fast retransmit and fast recovery.

The details of the algorithms differ slightly but in principle they intend to do the same thing [Ste07, pp. 94–100]. In slow start the congestion window is incremented at maximum by one path Maximum Transmission Unit (path MTU) per received acknowledgment of new data. In congestion avoidance the congestion window is incremented roughly by one MTU per round-trip time. Fast retransmit is entered after a data chunk is three times indicated missing. Congestion window is halved and a maximum of one MTU's worth of missing

data is retransmitted. Fast recovery is entered (if not already in fast recovery). In fast recovery the congestion window will not be changed. Upon entering the fast recovery, the highest transmission sequence number (TSN) of any data chunk that had been sent until that moment but was not yet acknowledged, will be the exit point for the fast recovery. When that TSN gets acknowledged, fast recovery exits. After exiting fast recovery SCTP continues with congestion avoidance.

One important difference between SCTP and TCP is that SCTP supports out-of-order deliveries [Ste07, p. 95]. This, however, does not create a big difference to the congestion control of the two. In the case of data received out-of-order, SCTP may deliver it to the application whereas TCP will keep it in the receiver buffer. Both TCP SACK and SCTP SACK will still report it as received in the acknowledgments, so there is no difference in that sense. However, when the data is delivered to the application and removed from the buffer the size of the receiver window may change, which in turn may affect the size of the congestion window (the congestion window can not be bigger than the receiver window).

Another difference is that SCTP supports partial reliability [Ram+04]. However, this is implemented as the sender sending a message to the receiver telling it to move forward its cumulative TSN point. From the congestion control point of view this is, in a sense, the same as if the receiver had received the actual data. Therefore, it does not seem to require any modifications to the congestion control algorithm to support this functionality.

Third, SCTP supports multi-homing [Ste07, p. 87]. The same endpoint can have multiple IP addresses within an SCTP association. However, in WebRTC, SCTP is on top of DTLS and DTLS does not support this functionality [Tüx+17, p. 5]. Therefore, the congestion control scheme becomes simpler and is similar to TCP in this regard.

In SCTP the whole association with multiple streams is under the same congestion control [Ste07, p. 94]. Therefore, if multiple Data Channels are opened their traffic is controlled as one [JLT15b, pp. 8–9]. However, the possible parallel video and audio streams of WebRTC run under a different protocol with their own congestion control. WebRTC might be developed to include a congestion control for the Data Channel that cooperates more optimally with the video and audio streams.

4.4 Applicability for Multiplayer Games

In this section we will review WebSocket, QUIC and WebRTC from the perspective of multiplayer games and how the protocols could be suitable as real-time game data transports.

WebSocket

WebSocket is a message-oriented protocol on top of a byte-oriented protocol, TCP [Gri13, p. 287]. WebSocket takes care of delivering distinct units of messages end-to-end. At the sender it splits the application message to one or more frames and at the receiver it reassembles the possibly multiple frames to form the original message [MF11, pp. 5–6, 27–38]. Each frame header introduces 2-14 bytes overhead.

Currently, based on The WebSocket API documentation at MDN Web Docs and a comment [Cow16] at stackoverflow.com it seems that the JavaScript WebSocket API does not provide a way for the application to control this message fragmentation. TCP protocol can also divide messages into different segments without the application control. The application thus sends and receives complete messages and is not aware of the fragmentation underneath. For game data transfer it could be a more intuitive concept to have a single game state message travel inside of a single network packet but this is something that can not be guaranteed with WebSocket.

The application data can be either binary or UTF-8 [MF11, p. 38]. The binary type is either “blob” or “arraybuffer” [MDN19]. The “arraybuffer” type is recommended to be used as a hint for the end host to keep the data in the memory instead of writing it to the disk [WHA20]. For a real-time application it may therefore be beneficial to consider using the “arraybuffer” binary type.

Minshall et al. [Min+00] discuss Nagle’s algorithm in TCP. Nagle’s algorithm is to prevent a sender from a situation where it unintentionally and continuously sends only a minimal size payload. This is also known as “silly window syndrome”. To avoid this situation, TCP tries to fill up a segment before sending it: TCP holds back sending less than Maximum Segment Size of data if any previous packets are unacknowledged. However, for a real-time application any delay can be undesired and therefore Nagle’s algorithm may not be preferred. As commented in [lpi16], Nagle’s algorithm for TCP is disabled by default in WebSocket.

WebSocket using TCP and TCP being a reliable and in-order protocol means that if there are gaps in packet sequence numbers at the receiver (due to lost or dropped packets) the data with higher sequence numbers will not be delivered to the application until TCP has retransmitted the missing data with lower sequence numbers and the data has reached the receiver. This HOL blocking can produce unacceptable delays for real-time games.

Mitigating the effects of HOL blocking can be attempted by establishing multiple parallel WebSocket connections. Many browsers allow maximum of six simultaneous connections to the same host [Gri13, p. 196]. For example, it might be possible to use parallel connections in the following way: A server sends updates to clients 30 times per second (about every 33 ms). The updates are spread over six connections to the same host. This makes 5 updates per second per connection. If one of the connections blocks, the five others still carry on. Assuming no delay variation, the blocking creates gaps of about 33 ms to the updates, at the rate of five times per second. This should be considerably easier to hide with the in-game delay-compensation methods than a delay caused by single connection HOL blocking that could reach to hundreds of milliseconds. The downside of this method is the complexity it introduces.

WebSocket is relatively easy to implement, especially with a JavaScript library Socket.io [Soc20] that uses WebSocket underneath but provides also other functionality and fallbacks for supporting the connection and data transfer. However, the fact that WebSocket uses TCP makes it suffer from delays that may be unacceptable for applications with stringent latency requirements.

QUIC and HTTP/3

It may be possible to exploit QUIC's stream multiplexing for transporting game state data. New streams are opened implicitly by setting a new stream number into the Stream ID header field of the frame that carries application data. In this same frame a FIN bit in the headers can be set to 1 which denotes that this frame is the last frame of the stream. Hence, a single game state update message could take a single QUIC packet containing a single frame. This solution is speculated in [No 16].

However, at the moment it seems that in web browsers QUIC can only be used through HTTP/3. This creates another layer and some overhead. Client would send its game data as a HTTP request. The server would send its game data as a server push. Each HTTP request creates a new client-initiated bidirectional QUIC stream and each server

push creates a new server-initiated unidirectional QUIC stream [Bis20, p. 12]. To read a received push message the client application has to request the particular message since the request has to match the push. This needs to be addressed in the design of the game.

A possible scheme could be that the client makes a HTTP request for game state updates to which the server sends a HTTP response containing a certain amount (for example, 1000) of PUSH_PROMISE frames. These frames reference the upcoming server game state updates about to be pushed to the client. When the client receives the set of PUSH_PROMISE frames it makes a request to all the data they refer to. The server then sends the pushes periodically as the game advances (for example, 60 times per second) and when they arrive the client gets to read them and update the game state. This scheme thus enables the server to immediately send and the client to immediately read when the data is ready. When the client is starting to run out of the push promises (for example, every 1000/60 seconds) it makes a new request for game state updates, and the loop continues.

The byte-overhead caused by HTTP headers is reduced by header compression. The header compression solution in HTTP/3 is called QPACK [KBF20]. It is similar to HPACK in HTTP/2 but it supports QUIC's unordered stream multiplexing. Headers are encoded at sender and decoded at the receiver. The encoder communicates with the decoder, and vice versa, by unidirectional streams. The header frames in different streams can arrive in arbitrary order and QPACK is able to decode them. However, since the ordering between streams is not preserved, a packet may reference a header that the encoder has not yet received and this would block that particular stream.

For a real-time game application the blocking of one stream by QPACK should not be a major drawback since it only affects the one game state update using that particular stream. The next update will come in another stream. However, this next update and all of the following ones might very well reference the same missing header. Then they all would become blocked until the header information arrives at the encoder.

Another issue that could cause delay is that QUIC may try to bundle frames before sending: it might wait for a short while if there is not enough data to fill a packet [IT20, pp. 87–88]. The draft says that this is an "implementation decision". It is unclear if an upper layer will have control over it or not. A draft regarding applicability of QUIC says that for a low latency application "it may be valuable to indicate to QUIC that all data should be send out immediately" but does not specify further [KT20, p. 9]. The draft for HTTP/3 does not comment on this either.

As an opposite to bundling, it also can not be known if a single application message will

get divided into separate network packets. QUIC is a byte-oriented protocol, it receives a continuous stream of data from the upper layer, cuts it at some point without notion of message boundaries and sends the bytes into the network. For a game data transport protocol an easier concept to reason with could be one where a single network packet was filled with a single game state message and sent immediately. However, this is not QUIC's intended use.

Ultimately, using QUIC and HTTP/3 as a substrate for networked multiplayer game traffic is sort of a hack or abuse of the protocol and without real-world implementation there is no assurance it will actually work.

Currently, QUIC and HTTP/3 are supported by default in Safari 14 browser for macOS 11 Big Sur operating system and can be user-enabled in Chrome and Firefox browsers [Dev20a]. For communication with Google servers Chrome is already using QUIC [Seu+19]. QUIC is also available in the Chromium software project [Buj+20].

WebRTC

WebRTC operates atop UDP and the Data Channel can be set as unreliable and unordered so it does not suffer from head-of-line blocking. Data Channel allows sending arbitrary binary messages. Therefore, WebRTC provides a solution that could be suitable for real-time game applications.

However, WebRTC is designed for peer-to-peer connections whereas networked multiplayer games often require a client-server architecture, for instance, to prevent cheating. Setting up a WebRTC connection where another peer would actually be a server may turn out to be quite laborious, as is noted in a blog post [Ho17]. The author of the blog manages to improvise a client-server WebRTC connection but illustratively comments that it would have taken about 10 lines of client code and about 20 lines of server code by using WebSocket, but by using WebRTC it took about 100 lines of client code and about 300 lines of server (C++) code [Ho17; Ho19]. Also, as another real-world example, the browser game Agar.io seem to be using WebSockets instead of WebRTC because WebRTC would be too complex [Mat16].

As mentioned in Chapter 3, the service behavior Assured Forwarding (AF) is, for example, for multimedia conferencing streams that can adjust their sending rate. The Expedited Forwarding (EF) is, for example, for interactive games that can not adjust their sending rate. WebRTC draft for DSCP markings [Jon+16, p. 6] recommends mappings of the

four Data Channel priority levels to the following DiffServ classes: below normal: CS1, normal: DF, high: AF11 and extra high AF21. Thus, the recommendations do not perhaps completely match the characteristics of inelastic real-time interactive game traffic.

An API is provided for manipulating the DSCP bits [Alv20] but currently it seems that this feature is not supported in any of the browsers [MDN20c]. It is also unclear if it would be allowed to set the DSCP to any arbitrary value or to only one of the four predefined values.

Since the client-server scheme is not what WebRTC is intended for, there may arise unexpected problems and the solutions may not work. This technique may therefore be prone to failures, at least until there are more real-world examples available of successful implementations. For peer-to-peer architectures WebRTC should be a fitting choice.

Currently, WebRTC is available on all major browsers except on Internet Explorer [Dev20b]. On the Edge browser, however, the RTCDataChannel is not supported and therefore Edge is also an unsuitable option.

The maturing of QUIC may affect WebRTC as well. Google is currently looking into possibly replacing SCTP with QUIC for WebRTC Data Channel [Ham19; BR20]. This may have implications on the message boundaries during transport since SCTP is a message-oriented protocol whereas QUIC is a byte-oriented protocol [Jos+18, p. 14]. However, this should not make a significant difference in its applicability as a game data transport.

5 Conclusions

In this thesis, we considered delay management for browser multiplayer games. As an overview on the problem space, we first looked into multiplayer game architectures, multiplayer game traffic characteristics and the specifics of using the web browser as a platform for multiplayer games. Then we went through the various delay sources that contribute to the overall network delay, and considered their effect on multiplayer games. Some prominent sources of network delay include signal propagation delay, queuing delay, packet loss recovery delay and head-of-line blocking delay. From these, our main focus was on queuing delays and head-of-line blocking delays.

As solutions for delay management in browser multiplayer games, we first briefly reviewed some in-game delay-compensation methods. In-game delay-compensation is an important tool in hiding the effects of network delay and for that reason was included in this thesis. Then we proceeded to take a closer look at the support a network can provide for multiplayer games. For this, we examined AQM, differentiated packet treatment and ECN. From the AQM algorithms, we chose RED, PIE, CoDel and FQ-CoDel to be studied in this thesis. RED is a well-known AQM algorithm whereas PIE, CoDel and FQ-CoDel are more recent proposals.

After considering the network support, we proceeded to take a look at some of the network protocols available in web browsers and to consider how they could satisfy the transport needs of delay-sensitive multiplayer games. The examined protocols were WebSocket, QUIC and WebRTC. From all the protocols available in web browsers these three appeared as the most reasonable options and were thus chosen for this thesis.

AQM and differentiated packet treatment were considered as the main tools for managing queuing delays, while ECN and the network protocols were considered as the main tools for managing head-of-line blocking delays. The findings were as follows.

In PIE and CoDel AQM algorithms the size of a packet does not affect the drop probability: a big or small packet is just as likely to be picked as the packet to be dropped or marked during congestion. This may not be optimal for small and frequent game state packets. RED provides an option where the packet size will, instead, affect the probability of the packet becoming dropped. This might suit game traffic better.

However, RED's parametrization regarding suitable queue size thresholds is problematic. Extensions such as ARED have been proposed to mitigate the problem. In a performance evaluation, however, ARED has performed slightly worse than PIE or CoDel. CoDel's default drop mode interval of 100 ms has been recommended to be reduced for improved performance.

Multi-queue AQM algorithm FQ-CoDel provides good flow separation and favors thin or short flows. This eliminates the problem in single-queue schemes where small packets would get unfair treatment, and thus FQ-CoDel could be beneficial for multiplayer game traffic. On the other hand, multi-queue schemes can be problematic in combination with the lower-level queuing that exists in network devices. The CAKE algorithm seeks to alleviate this problem by using set-associative hashing for the queues.

Regarding DiffServ, the CS4 class and the AF4x class could be considered suitable for multiplayer game traffic. CS4 could provide the fastest forwarding since it is meant for latency-sensitive interactive applications that can not react to congestion, and is expected to be provided enough bandwidth from the network to avoid packet drops. The AF4x class is meant for real-time interactive multimedia applications such as video conferencing and could also be used for multiplayer games. However, there are no guarantees that network operators adhere to the DiffServ recommendations for CS4 or AF4x and thus for some cases the default best-effort class may be the best option.

The use of ECN can reduce packet losses and associated delays such as HOL blocking. Delays occur with reliable transports where packet retransmission is necessary in the event of packet loss. With non-reliable transport, packet loss recovery delays and HOL blocking can be avoided but data may be lost, instead. For multiplayer games, packet losses with non-reliable transport can create gaps in the sequence of update message and have an effect similar to network delay. When ECN-capable and non-ECN-capable traffic share a highly congested link all the ECN-capable packets may become ECN-CE flagged and forwarded while all the non-ECN-capable packets may become dropped, which can be considered as an unfair situation for the non-ECN-capable flows. Therefore, in general and for multiplayer games, the use of ECN can be beneficial.

In order for ECN to work properly it needs support from both end-hosts and from the intermediaries on the network path. ECN is originally designed to be used with TCP and IP. If ECN is to be used with UDP and IP, instead, the ECN communication between end-hosts needs to be implemented at an upper layer such as on the application layer. However, access of IP layer ECN bits for application layer may be uncertain on some

operating systems.

On the browser protocols, WebSocket provides a simple API and is meant for transferring arbitrary data such as game data between a client and a server. However, it uses TCP as a substrate and suffers from HOL blocking which may create delays unacceptable for fast paced multiplayer games. Opening multiple parallel connections could provide latency benefits but with added complexity on the design.

QUIC uses UDP and avoids HOL blocking between streams. However, QUIC is not directly exposed for applications in browsers but needs to be utilized via HTTP/3. Thus, QUIC is not intended for game traffic which makes the game scheme slightly complex, involves overhead and ultimately provides no guarantees that it will actually work.

WebRTC uses UDP, can avoid HOL blocking, and includes a data channel that seems suitable for game data transfer. For peer-to-peer architectures WebRTC could be a fitting choice. For client-server usage WebRTC involves complexity that may hinder its utilization.

Future studies could include experimenting with QUIC and WebRTC to receive actual first-hand information on how they would suit as a transport for game data. QUIC is already available on many browsers either by default or optionally, requiring user to enable it. WebRTC is available on all major browsers except on Internet Explorer.

As a summarized answer to our research question ”what feasible networking solutions exist for browser multiplayer games?” we conclude the following. Currently, WebRTC and FQ-CoDel seem as promising options. WebRTC DataChannel avoids HOL blocking delays and it can be used for sending arbitrary application data. FQ-CoDel provides flow separation that is able to prevent queue-building bulk transfers from notably hampering latency-sensitive flows. However, WebRTC and FQ-CoDel both involve some complexities and open questions: how does WebRTC fit for client-server architectures and how do network devices handle the multiple queues in FQ-CoDel? These questions are left for future studies and experiments.

Bibliography

[AC17] G. Armitage and R. Collom. "Benefits of FlowQueue-Based Active Queue Management for Interactive Online Games". In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. Vancouver, British Columbia, Canada, 2017, pp. 1–9.

[ACB06] G. Armitage, M. Claypool, and P. Branch. *Networking and Online Games*. John Wiley & Sons Ltd, 2006. ISBN: 9780470018576.

[Ali+10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data Center TCP (DCTCP)". In: *SIGCOMM Computer Communication Review* 40.4 (Aug. 2010), pp. 63–74. ISSN: 0146-4833. DOI: 10.1145/1851275.1851192. URL: https://doi.org/10.1145/1851275.1851192.

[Alv11] H. Alvestrand. *Google release of WebRTC source code. W3C Mailing lists*. 2011. URL: https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html (visited on 10/15/2020).

[Alv17] H. T. Alvestrand. *Overview: Real Time Protocols for Browser-based Applications*. Internet-Draft draft-ietf-rtcweb-overview-19. Work in Progress. Internet Engineering Task Force, Nov. 2017. 24 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-rtcweb-overview-19.

[Alv20] H. Alvestrand. *WebRTC Priority Control API. W3C Working Draft, 22 September 2020*. 2020. URL: https://www.w3.org/TR/2020/WD-webrtc-priority-20200922/ (visited on 10/15/2020).

[Ant15] T. Anttalainen. *Introduction to Communication Networks*. Boston: Artech House, 2015. ISBN: 1608077624.

[Aur+20] D. Aureli, A. Cianfrani, A. Diamanti, J. M. Sanchez Vilchez, and S. Secci. "Going Beyond DiffServ in IP Traffic Classification". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. Budapest, Hungary, 2020, pp. 1–6.

[Bak+98] F. Baker, D. L. Black, K. Nichols, and S. L. Blake. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC 2474. Dec. 1998. DOI: 10.17487/RFC2474. URL: https://rfc-editor.org/rfc/rfc2474.txt.

[BBC06] F. Baker, J. Babiarz, and K. H. Chan. *Configuration Guidelines for DiffServ Service Classes*. RFC 4594. Aug. 2006. DOI: 10.17487/RFC4594. URL: https://rfc-editor.org/rfc/rfc4594.txt.

[BCS94] R. T. Braden, D. D. D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview*. RFC 1633. June 1994. DOI: 10.17487/RFC1633. URL: https://rfc-editor.org/rfc/rfc1633.txt.

[Ber01] Y. Bernier. "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization". In: *Proceedings of the Game Developers Conference*. San Jose, California, USA, 2001. URL: https://web.cs.wpi.edu/~claypool/courses/4513-B03/papers/games/bernier.pdf.

[BF10] M. Bredel and M. Fidler. "A Measurement Study Regarding Quality of Service and Its Impact on Multiplayer Online Games". In: *2010 9th Annual Workshop on Network and Systems Support for Games*. Taipei, Taiwan, 2010, pp. 1–6.

[BF15] F. Baker and G. Fairhurst. *IETF Recommendations Regarding Active Queue Management*. RFC 7567. July 2015. DOI: 10.17487/RFC7567. URL: https://rfc-editor.org/rfc/rfc7567.txt.

[Bha15] U. N. Bhat. *An Introduction to Queueing Theory: Modeling and Analysis in Applications*. 2nd ed. 2015. Statistics for Industry and Technology. Boston, MA: Birkhäuser Boston, 2015. ISBN: 9780817684204.

[Bis20] M. Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Internet-Draft draft-ietf-quic-http-31. Work in Progress. Internet Engineering Task Force, Sept. 2020. 72 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-31.

[BJ15] D. L. Black and P. Jones. *Differentiated Services (Diffserv) and Real-Time Communication*. RFC 7657. Nov. 2015. DOI: 10.17487/RFC7657. URL: https://rfc-editor.org/rfc/rfc7657.txt.

[BPA09] E. Blanton, D. V. Paxson, and M. Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681. URL: https://rfc-editor.org/rfc/rfc5681.txt.

[BPT15] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540. URL: https://rfc-editor.org/rfc/rfc7540.txt.

[BR20] P. T. and Bernard Aboba and R. Raymond. *QUIC API for Peer-to-peer Connections. Draft Community Group Report 21 January 2020*. 2020. URL: https://w3c.github.io/webrtc-quic/ (visited on 10/15/2020).

[Bra+97] R. T. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*. RFC 2205. Sept. 1997. DOI: 10.17487/RFC2205. URL: https://rfc-editor.org/rfc/rfc2205.txt.

[Bri+16] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl. "Reducing Internet Latency: A Survey of Techniques and Their Merits". In: *IEEE Communications Surveys & Tutorials* 18.3 (2016), pp. 2149–2196.

[Buj+20] A. Bujari, C. E. Palazzi, G. Quadrio, and D. Ronzani. "Emerging Interactive Applications over QUIC". In: *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*. Las Vegas, Nevada, USA, 2020, pp. 1–4.

[Cab14] CableLabs®. *Active Queue Management In DOCSIS 3.x Cable modems*. Cable Television Laboratories, Inc. ("CableLabs®"), 2014. URL: https://www.cablelabs.com/wp-content/uploads/2014/06/DOCSIS-AQM_May2014.pdf.

[CC06] M. Claypool and K. Claypool. "Latency and Player Actions in Online Games". In: *Communications of the ACM* 49.11 (Nov. 2006), pp. 40–45. ISSN: 0001-0782. DOI: 10.1145/1167838.1167860. URL: https://doi.org/10.1145/1167838.1167860.

[CCG19] M. Claypool, A. Cockburn, and C. Gutwin. "Game Input with Delay: Moving Target Selection Parameters". In: *Proceedings of the 10th ACM Multimedia Systems Conference*. MMSys '19. Amherst, Massachusetts, USA: Association for Computing Machinery, 2019, pp. 25–35. ISBN: 9781450362979. DOI: 10.1145/3304109.3306232. URL: https://doi.org/10.1145/3304109.3306232.

[CDM06] B. Carrig, D. Denieffe, and J. Murphy. “A Relative Delay Minimization Scheme for Multiplayer Gaming in Differentiated Services Networks”. In: *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames ’06. Singapore: Association for Computing Machinery, 2006, 36–es. ISBN: 1595935894. DOI: 10.1145/1230040.1230046. URL: https://doi.org/10.1145/1230040.1230046.

[Che17] C. Chen. “Measuring the State of ECN on the Internet”. In: *2017 International Conference on Information and Communication Technology Convergence (ICTC)*. Jeju, South Korea, 2017, pp. 172–177.

[CI12] X. Che and B. Ip. “Packet-Level Traffic Analysis of Online Games from the Genre Characteristics Perspective”. In: *Elsevier Journal of Network and Computer Applications* 35.1 (Jan. 2012), pp. 240–252. DOI: 10.1016/j.jnca.2011.08.005.

[Cla05] M. Claypool. “The Effect of Latency on User Performance in Real-Time Strategy Games”. In: *Computer Networks* 49.1 (Sept. 2005), pp. 52–70. ISSN: 1389-1286.

[Cow16] M. Cowie. *How to trigger websocket frame fragmentation from the client? answered Apr 3 ’16 at 15:48*. 2016. URL: https://stackoverflow.com/questions/36387499/how-to-trigger-websocket-frame-fragmentation-from-the-client (visited on 10/15/2020).

[Cox20] B. Cox. *How 1500 bytes became the MTU of the internet*. 2020. URL: https://blog.benjojo.co.uk/post/why-is-ethernet-mtu-1500 (visited on 10/15/2020).

[CSF18] A. Custura, R. Secchi, and G. Fairhurst. “Exploring DSCP Modification Pathologies in the Internet”. In: *Computer Communications* 127 (2018), pp. 86–94. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2018.05.016. URL: http://www.sciencedirect.com/science/article/pii/S0140366417312835.

[DB20] M. Duke and N. Banks. *QUIC-LB: Generating Routable QUIC Connection IDs*. Internet-Draft draft-ietf-quic-load-balancers-04. Work in Progress. Internet Engineering Task Force, Aug. 2020. 30 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-load-balancers-04.

[Dev20a] A. Deveria. *Can I Use QUIC*. Can I Use. 2020. URL: https://caniuse.com/#search=quic (visited on 10/15/2020).

[Dev20b] A. Deveria. *Can I Use WebRTC*. Can I Use. 2020. URL: https://caniuse.com/?search=webrtc (visited on 10/18/2020).

[DH17] S. E. Deering and B. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. July 2017. DOI: 10.17487/RFC8200. URL: https://rfc-editor.org/rfc/rfc8200.txt.

[Dio+00] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. "Deployment Issues for the IP Multicast Service and Architecture". In: *IEEE Network* 14.1 (2000), pp. 78–88.

[DWW05] M. Dick, O. Wellnitz, and L. Wolf. "Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games". In: *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames '05. Hawthorne, New York, USA: Association for Computing Machinery, 2005, pp. 1–7. ISBN: 1595931562. DOI: 10.1145/1103599.1103624. URL: https://doi.org/10.1145/1103599.1103624.

[EFS17] L. Eggert, G. Fairhurst, and G. Shepherd. *UDP Usage Guidelines*. RFC 8085. Mar. 2017. DOI: 10.17487/RFC8085. URL: https://rfc-editor.org/rfc/rfc8085.txt.

[ER20] E. J. Etemad and F. Rivoal. *World Wide Web Consortium Process Document, 01 March 2019*. 2020. URL: https://www.w3.org/2020/Process-20200915/ (visited on 10/15/2020).

[Esp05] N. Esposito. "A Short and Simple Definition of What a Videogame Is". In: *Proceedings of DiGRA 2005 Conference: Changing Views – Worlds in Play*. Vancouver, British Columbia, Canada: Digital Games Research Association DiGRA, Jan. 2005.

[Fen+99] W. Feng, D. D. Kandlur, D. Saha, and K. G. Shin. "A self-configuring RED gateway". In: *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*. Vol. 3. New York, New York, USA, 1999, 1320–1328 vol.3.

[FF96] K. Fall and S. Floyd. "Simulation-Based Comparisons of Tahoe, Reno and SACK TCP". In: *SIGCOMM Computer Communication Review* 26.3 (July 1996), pp. 5–21. ISSN: 0146-4833. DOI: 10.1145/235160.235162. URL: https://doi.org/10.1145/235160.235162.

[FGS01] S. Floyd, R. Gummadi, and S. Shenker. "Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management". In: (Sept. 2001). URL: http://www.icir.org/floyd/papers/adaptiveRed.pdf.

[Fie14a] G. Fiedler. *Deterministic Lockstep*. 2014. URL: https://gafferongames.com/post/deterministic_lockstep/ (visited on 10/15/2020).

[Fie14b] G. Fiedler. *Snapshot Interpolation*. 2014. URL: https://gafferongames.com/post/snapshot_interpolation/ (visited on 10/15/2020).

[Fie15a] G. Fiedler. *Snapshot Compression*. 2015. URL: https://gafferongames.com/post/snapshot_compression/ (visited on 10/15/2020).

[Fie15b] G. Fiedler. *State Synchronization*. 2015. URL: https://gafferongames.com/post/state_synchronization/ (visited on 10/15/2020).

[FJ93] S. Floyd and V. Jacobson. *Random Early Detection gateways for Congestion Avoidance*. Lawrence Berkeley Laboratory, 1993.

[Flo+96] S. Floyd, J. Mahdavi, M. Mathis, and D. A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. Oct. 1996. DOI: 10.17487/RFC2018. URL: https://rfc-editor.org/rfc/rfc2018.txt.

[FR14] R. T. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. June 2014. DOI: 10.17487/RFC7231. URL: https://rfc-editor.org/rfc/rfc7231.txt.

[FRB01] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: 10.17487/RFC3168. URL: https://rfc-editor.org/rfc/rfc3168.txt.

[GN11] J. Gettys and K. Nichols. "Bufferbloat: Dark Buffers in the Internet". In: *Queue* 9.11 (Nov. 2011), pp. 40–54. ISSN: 1542-7730. DOI: 10.1145/2063166.2071893. URL: https://doi.org/10.1145/2063166.2071893.

[Goo19] Google. *WebRTC - Support - Standardization*. 2019. URL: https://webrtc.org/support/standardization (visited on 10/15/2020).

[GR16] A. Gertsiy and S. Rudyk. “Analysis of Quality of Service Parameters in IP-Networks”. In: *2016 Third International Scientific-Practical Conference Problems of Infocommunications Science and Technology (PIC S&T)*. Kharkiv, Ukraine, 2016, pp. 75–77.

[Gre18] J. Gregory. *Game Engine Architecture*. Third edition. Boca Raton: Chapman and Hall/CRC, 2018. ISBN: 9781351974288.

[Gri13] I. Grigorik. *High-Performance Browser Networking*. O’Reilly Media, Inc., 2013. ISBN: 9781449344764.

[Ham19] S. Hampson. *RTCQuicTransport Coming to an Origin Trial Near You (Chrome 73)*. 2019. URL: https://developers.google.com/web/updates/2019/01/rtcquictransport-api (visited on 10/15/2020).

[HH07] D. Harris and S. Harris. *Digital Design and Computer Architecture: From Gates to Processors*. Burlington: Elsevier Science & Technology, 2007. ISBN: 9780123704979.

[Ho17] B. Ho. *A comprehensive dive into WebRTC for client-server web games. beep boop a game development blog*. 2017. URL: http://blog.brkho.com/2017/03/15/dive-into-client-server-web-games-webrtc/ (visited on 10/15/2020).

[Ho19] B. Ho. *An example for using WebRTC to communicate between a JavaScript client and a C++ server. Commit 39f6890*. 2019. URL: https://github.com/brkho/client-server-webrtc-example (visited on 10/15/2020).

[Høi+18] T. Høiland-Jørgensen, P. McKenney, D. Täht, J. Gettys, and E. Dumazet. *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*. RFC 8290. Jan. 2018. DOI: 10.17487/RFC8290. URL: https://rfc-editor.org/rfc/rfc8290.txt.

[Hor84] C. Hornig. *A Standard for the Transmission of IP Datagrams over Ethernet Networks*. RFC 894. Apr. 1984. DOI: 10.17487/RFC0894. URL: https://rfc-editor.org/rfc/rfc894.txt.

[How+14] E. Howard, C. Cooper, M. P. Wittie, S. Swinford, and Q. Yang. “Cascading Impact of Lag on Quality of Experience in Cooperative Multiplayer Games”. In: *2014 13th Annual Workshop on Network and Systems Support for Games*. Nagoya, Japan, 2014, pp. 1–6.

[HR13] D. Hayes and D. Ros. “Delay-based Congestion Control for Low Latency”. In: (Sept. 2013). URL: https://www.internetsociety.org/wp-content/uploads/2013/09/17_delay_cc_pos-v2.pdf (visited on 10/15/2020).

[HTM18] T. Høiland-Jørgensen, D. Täht, and J. Morton. “Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways”. In: *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. Washington, D.C., USA, 2018, pp. 37–42.

[IET20a] IETF. *Concluded Working Groups*. Mar. 2020. URL: https://tools.ietf.org/wg/concluded (visited on 10/15/2020).

[IET20b] IETF. *Rmcat Status Pages, RTP Media Congestion Avoidance Techniques (Active WG)*. 2020. URL: https://tools.ietf.org/wg/rmcat/index.pyht?sort=3&reverse=0 (visited on 10/15/2020).

[IS20] J. Iyengar and I. Swett. *QUIC Loss Detection and Congestion Control*. Internet-Draft draft-ietf-quic-recovery-31. Work in Progress. Internet Engineering Task Force, Sept. 2020. 51 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-31.

[IT20] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-31. Work in Progress. Internet Engineering Task Force, Sept. 2020. 199 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-31.

[Ivk+15] Z. Ivkovic, I. Stavness, C. Gutwin, and S. Sutcliffe. “Quantifying and Mitigating the Negative Effects of Local Latencies on Aiming in 3D Shooter Games”. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI ’15. Seoul, Republic of Korea: Association for Computing Machinery, 2015, pp. 135–144. ISBN: 9781450331456. DOI: 10.1145/2702123.2702432. URL: https://doi.org/10.1145/2702123.2702432.

[IWG13] S. Islam, M. Welzl, and S. Gjessing. “One Control to Rule Them All: Coupled Congestion Control for RTP Media”. In: *Packet Video Workshop*. San Jose, California, USA: University of Oslo, 2013. URL: http://heim.ifi.uio.no/michawe/research/publications/pv2013-fse-poster-final.pdf.

[JBB20] C. Jennings, H. Boström, and J.-I. Bruaroey. *WebRTC 1.0: Real-time Communication Between Browsers. W3C Candidate Recommendation 08 October*

2020. 2020. URL: https://www.w3.org/TR/2020/CR-webrtc-20200903/ (visited on 10/15/2020).

[JLT15a] R. Jesup, S. Loreto, and M. Tüxen. *WebRTC Data Channel Establishment Protocol*. Internet-Draft draft-ietf-rtcweb-data-protocol-09. Work in Progress. Internet Engineering Task Force, Jan. 2015. 13 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-rtcweb-data-protocol-09.

[JLT15b] R. Jesup, S. Loreto, and M. Tüxen. *WebRTC Data Channels*. Internet-Draft draft-ietf-rtcweb-data-channel-13. Work in Progress. Internet Engineering Task Force, Jan. 2015. 16 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-rtcweb-data-channel-13.

[Jon+16] P. Jones, S. Dhesikan, C. Jennings, and D. Druta. *DSCP Packet Markings for WebRTC QoS*. Internet-Draft draft-ietf-tsvwg-rtcweb-qos-18. Work in Progress. Internet Engineering Task Force, Aug. 2016. 11 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-rtcweb-qos-18.

[Jon19] A. Jones. *The Most Popular Video Game Genres in 2019*. 2019. URL: https://invisioncommunity.co.uk/the-most-popular-video-game-genres-in-2019/ (visited on 10/15/2020).

[Jos+18] A. Joseph, T. Li, Z. He, Y. Cui, and L. Zhang. *A Comparison Between SCTP and QUIC*. Internet-Draft draft-joseph-quic-comparison-quic-sctp-00. Work in Progress. Internet Engineering Task Force, Mar. 2018. 24 pp. URL: https://datatracker.ietf.org/doc/html/draft-joseph-quic-comparison-quic-sctp-00.

[KBF20] C. Krasic, M. Bishop, and A. Frindell. *QPACK: Header Compression for HTTP/3*. Internet-Draft draft-ietf-quic-qpack-18. Work in Progress. Internet Engineering Task Force, Sept. 2020. 49 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-qpack-18.

[KD11] P. Kanuparthy and C. Dovrolis. "ShaperProbe: End-to-End Detection of ISP Traffic Shaping Using Active Methods". In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement*. IMC '11. Berlin, Germany: Association for Computing Machinery, 2011, pp. 473–482. ISBN: 9781450310130. DOI: 10.1145/2068816.2068860. URL: https://doi.org/10.1145/2068816.2068860.

[Ken02] T. Kenyon. *High-Performance Data Network Design: Design Techniques and Tools*. Boston: Digital Press, 2002. ISBN: 9780585457475.

[KHR18] A. Keränen, C. Holmberg, and J. Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. July 2018. DOI: 10.17487/RFC8445. URL: https://rfc-editor.org/rfc/rfc8445.txt.

[Kob06] K. Kobayashi. "Transmission Timer Approach for Rate Based Pacing TCP with Hardware Support". In: (Feb. 2006). URL: https://www.researchgate.net/publication/228873177_Transmission_timer_approach_for_rate_based_pacing_TCP_with_hardware_support (visited on 10/15/2020).

[KRW14] N. Khademi, D. Ros, and M. Welzl. "The New AQM Kids on the Block: An Experimental Evaluation of CoDel and PIE". In: *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Toronto, Ontario, Canada, 2014, pp. 85–90.

[KT20] M. Kühlewind and B. Trammell. *Applicability of the QUIC Transport Protocol*. Internet-Draft draft-ietf-quic-applicability-07. Work in Progress. Internet Engineering Task Force, July 2020. 17 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-applicability-07.

[LC18] S. W. K. Lee and R. K. C. Chang. "Enhancing the Experience of Multiplayer Shooter Games via Advanced Lag Compensation". In: *Proceedings of the 9th ACM Multimedia Systems Conference*. MMSys '18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 284–293. ISBN: 9781450351928. DOI: 10.1145/3204949.3204971. URL: https://doi.org/10.1145/3204949.3204971.

[LJS17] J. Luo, J. Jin, and F. Shan. "Standardization of Low-Latency TCP with Explicit Congestion Notification: A Survey". In: *IEEE Internet Computing* 21.1 (2017), pp. 48–55.

[lpi16] lpinca. *[request] Disable Nagle's algorithm #791. lpinca commented on 21 Jul 2016, edited*. 2016. URL: https://github.com/websockets/ws/issues/791 (visited on 10/15/2020).

[Lu20] M. Lu. *The 50 Biggest Video Game Franchises by Total Revenue*. 2020. URL: https://www.visualcapitalist.com/50-biggest-video-game-franchises-revenue/ (visited on 10/15/2020).

[LZC05] Lei Liang, Zhili Sun, and H. Cruickshank. “Relative QoS Optimization for Multiparty Online Gaming in DiffServ Networks”. In: *IEEE Communications Magazine* 43.5 (2005), pp. 75–83.

[Mad18] S. Madhav. *Game Programming in C++. Creating 3D Games*. Addison-Wesley Professional, 2018. ISBN: 9780134598185.

[Mat+08] P. Matthews, J. Rosenberg, D. Wing, and R. Mahy. *Session Traversal Utilities for NAT (STUN)*. RFC 5389. Oct. 2008. DOI: 10.17487/RFC5389. URL: https://rfc-editor.org/rfc/rfc5389.txt.

[Mat16] Matheus28. *WebRTC: the future of web games (getkey.eu). Matheus28 on Dec 27, 2016*. 2016. URL: https://news.ycombinator.com/item?id=13264952 (visited on 10/15/2020).

[May+99] M. May, J. Bolot, C. Diot, and B. Lyles. “Reasons Not to Deploy RED”. In: *1999 Seventh International Workshop on Quality of Service. IWQoS’99. (Cat. No.98EX354)*. London, United Kingdom, 1999, pp. 260–262.

[MCB08] A. Malik Khan, S. Chabridon, and A. Beugnard. “A Dynamic Approach to Consistency Management for Mobile Multiplayer Games”. In: *Proceedings of the 8th International Conference on New Technologies in Distributed Systems*. NOTERE ’08. Lyon, France: Association for Computing Machinery, 2008. ISBN: 9781595939371. DOI: 10.1145/1416729.1416783. URL: https://doi.org/10.1145/1416729.1416783.

[McM18] P. McManus. *Bootstrapping WebSockets with HTTP/2*. RFC 8441. Sept. 2018. DOI: 10.17487/RFC8441. URL: https://rfc-editor.org/rfc/rfc8441.txt.

[MDN19] MDN contributors. *WebSocket.send(). MDN web docs*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket/send (visited on 10/15/2020).

[MDN20a] MDN contributors. *Concurrency model and the event loop. MDN web docs*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop (visited on 10/15/2020).

[MDN20b] MDN contributors. *Cross-Origin Resource Sharing (CORS). MDN web docs*. Feb. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (visited on 10/15/2020).

[MDN20c] MDN contributors. *RTCDataChannel. MDN web docs*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel (visited on 10/15/2020).

[MDN20d] MDN contributors. *SharedArrayBuffer. MDN web docs*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer (visited on 10/15/2020).

[MDN20e] MDN contributors. *Using Web Workers. MDN web docs*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (visited on 10/15/2020).

[MDN20f] MDN contributors. *Window.requestAnimationFrame(). MDN web docs*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame (visited on 10/15/2020).

[MF11] A. Melnikov and I. Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://rfc-editor.org/rfc/rfc6455.txt.

[Mic19a] Microsoft. *IPPROTO_IP Socket Options. Docs / Windows / Windows Sockets 2 / Winsock Reference / Socket Options / IPPROTO_IP Socket Options*. Microsoft. 2019. URL: https://docs.microsoft.com/en-gb/windows/win32/winsock/ipproto-ip-socket-options (visited on 10/15/2020).

[Mic19b] Microsoft. *IPPROTO_IPV6 Socket Options. Docs / Windows / Windows Sockets 2 / Winsock Reference / Socket Options / IPPROTO_IPV6 Socket Options*. Microsoft. 2019. URL: https://docs.microsoft.com/en-gb/windows/win32/winsock/ipproto-ipv6-socket-options (visited on 10/15/2020).

[Min+00] G. Minshall, Y. Saito, J. C. Mogul, and B. Verghese. "Application Performance Pitfalls and TCP's Nagle Algorithm". In: *SIGMETRICS Performance Evaluation Review* 27.4 (Mar. 2000), pp. 36–44. ISSN: 0163-5999. DOI: 10.1145/346000.346012. URL: https://doi.org/10.1145/346000.346012.

[Moc87] P. Mockapetris. *Domain Names - Concepts and Facilities*. RFC 1034. Nov. 1987. DOI: 10.17487/RFC1034. URL: https://rfc-editor.org/rfc/rfc1034.txt.

[MR08] M. Maier and M. Reisslein. "Trends in Optical Switching Techniques: A Short Survey". In: *IEEE Network* 22.6 (2008), pp. 42–47.

[MR10] D. McGrew and E. Rescorla. *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)*. RFC 5764. May 2010. DOI: 10.17487/RFC5764. URL: https://rfc-editor.org/rfc/rfc5764.txt.

[MR16] D. Madhuri and P. C. Reddy. "Performance Comparison of TCP, UDP and SCTP in a Wired Network". In: *2016 International Conference on Communication and Electronics Systems (ICCES)*. Coimbatore, India, 2016, pp. 1–6.

[MRM10] P. Matthews, J. Rosenberg, and R. Mahy. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 5766. Apr. 2010. DOI: 10.17487/RFC5766. URL: https://rfc-editor.org/rfc/rfc5766.txt.

[MS00] M. H. MacGregor and W. Shi. "Deficits for Bursty Latency-Critical Flows: DRR++". In: *Proceedings IEEE International Conference on Networks 2000 (ICON 2000). Networking Trends and Challenges in the New Millennium*. Singapore, 2000, pp. 287–293.

[Nic+18] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar. *Controlled Delay Active Queue Management*. RFC 8289. Jan. 2018. DOI: 10.17487/RFC8289. URL: https://rfc-editor.org/rfc/rfc8289.txt.

[No 16] "No Bugs" Hare. *UDP for games – security (encryption and DDoS protection)*. 2016. URL: http://ithare.com/udp-for-games-security-encryption-and-ddos-protection/ (visited on 10/15/2020).

[Nor+04] K. Norrman, D. McGrew, M. Naslund, E. Carrara, and M. Baugher. *The Secure Real-time Transport Protocol (SRTP)*. RFC 3711. Mar. 2004. DOI: 10.17487/RFC3711. URL: https://rfc-editor.org/rfc/rfc3711.txt.

[Nys14] R. Nystrom. *Game programming patterns*. Poland, Wrocław: Genever Benning, 2014. ISBN: 9780990582908.

[Oli18] J. Olivetti. *Perfect Ten: MMOs with different (camera) perspectives*. 2018. URL: https://massivelyop.com/2018/05/02/perfect-ten-mmos-with-different-camera-perspectives/ (visited on 10/15/2020).

[Pan+17] R. Pan, P. Natarajan, F. Baker, and G. White. *Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem*. RFC 8033. Feb. 2017. DOI: 10.17487/RFC8033. URL: https://rfc-editor.org/rfc/rfc8033.txt.

[PAS99] D. V. Paxson, M. Allman, and W. R. Stevens. *TCP Congestion Control*. RFC 2581. Apr. 1999. DOI: 10.17487/RFC2581. URL: https://rfc-editor.org/rfc/rfc2581.txt.

[PHJ06] C. Perkins, M. J. Handley, and V. Jacobson. *SDP: Session Description Protocol*. RFC 4566. July 2006. DOI: 10.17487/RFC4566. URL: https://rfc-editor.org/rfc/rfc4566.txt.

[Pos80] J. Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://rfc-editor.org/rfc/rfc768.txt.

[Pos81] J. Postel. *Transmission Control Protocol*. RFC 793. Sept. 1981. DOI: 10.17487/RFC0793. URL: https://rfc-editor.org/rfc/rfc793.txt.

[PW02] L. Pantel and L. C. Wolf. "On the Impact of Delay on Real-Time Multiplayer Games". In: *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV '02. Miami, Florida, USA: Association for Computing Machinery, 2002, pp. 23–29. ISBN: 1581135122. DOI: 10.1145/507670.507674. URL: https://doi.org/10.1145/507670.507674.

[Qua+04] P. Quax, P. Monsieurs, W. Lamotte, D. De Vleeschauwer, and N. Degrande. "Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game". In: *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames '04. Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 152–156. ISBN: 158113942X. DOI: 10.1145/1016540.1016557. URL: https://doi.org/10.1145/1016540.1016557.

[QUI19] QUICWG. *ECN in QUIC. quicwg/base-drafts, edited by Dmitri Tikhonov 15 Feb 2019*. 2019. URL: https://github.com/quicwg/base-drafts/wiki/ECN-in-QUIC (visited on 10/15/2020).

[Ram+04] D. M. A. Ramalho, Q. Xie, R. R. Stewart, M. Tüxen, and P. Conrad. *Stream Control Transmission Protocol (SCTP) Partial Reliability Extension*. RFC

3758. May 2004. DOI: 10.17487/RFC3758. URL: https://rfc-editor.org/rfc/rfc3758.txt.

[Ram20] G. Ramakrishnan. *FQ-PIE-for-Linux-Kernel. Commit 9e16f6c*. 2020. URL: https://github.com/gautamramk/FQ-PIE-for-Linux-Kernel (visited on 10/15/2020).

[Res19] E. Rescorla. *WebRTC Security Architecture*. Internet-Draft draft-ietf-rtcweb-security-arch-20. Work in Progress. Internet Engineering Task Force, July 2019. 43 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-rtcweb-security-arch-20.

[RFP08] M. Roccetti, S. Ferretti, and C. E. Palazzi. "The Brave New World of Multiplayer Online Games: Synchronization Issues with Smart Solutions". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. Orlando, Florida, USA, 2008, pp. 587–592.

[rje16] rjesup@mozilla.com. *What is RMCAT congestion control, and how will it affect WebRTC?* 2016. URL: https://blog.mozilla.org/webrtc/what-is-rmcat-congestion-control/ (visited on 10/15/2020).

[RM12] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347. URL: https://rfc-editor.org/rfc/rfc6347.txt.

[RP15] K. Raaen and A. Petlund. "How Much Delay is There Really in Current Games?" In: *Proceedings of the 6th ACM Multimedia Systems Conference*. MMSys '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 89–92. ISBN: 9781450333511. DOI: 10.1145/2713168.2713188. URL: https://doi.org/10.1145/2713168.2713188.

[RSR08] M. Ries, P. Svoboda, and M. Rupp. "Empirical Study of Subjective Quality for Massive Multiplayer Games". In: *2008 15th International Conference on Systems, Signals and Image Processing*. Bratislava, Slovakia, 2008, pp. 181–184.

[Sav+14] C. Savery, N. Graham, C. Gutwin, and M. Brown. "The Effects of Consistency Maintenance Methods on Player Experience and Performance in Networked Games". In: *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW '14. Balti-

more, Maryland, USA: Association for Computing Machinery, 2014, pp. 1344–1355. ISBN: 9781450325400. DOI: 10.1145/2531602.2531616. URL: https://doi.org/10.1145/2531602.2531616.

[Sch+03] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. July 2003. DOI: 10.17487/RFC3550. URL: https://rfc-editor.org/rfc/rfc3550.txt.

[Seu+19] M. Seufert, R. Schatz, N. Wehner, and P. Casas. "QUICker or not? -an Empirical Analysis of QUIC vs TCP for Video Streaming QoE Provisioning". In: *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. Paris, France, 2019, pp. 7–12.

[SG14] C. Savery and N. Graham. "Reducing the Negative Effects of Inconsistencies in Networked Games". In: *Proceedings of the First ACM SIGCHI Annual Symposium on Computer-Human Interaction in Play*. CHI PLAY '14. Toronto, Ontario, Canada: Association for Computing Machinery, 2014, pp. 237–246. ISBN: 9781450330145. DOI: 10.1145/2658537.2658539. URL: https://doi.org/10.1145/2658537.2658539.

[Sil15] R. Silveira. *Multiplayer Game Development with HTML5*. Community Experience Distilled. Packt Publishing, 2015. ISBN: 9781785283109.

[SNH04] J. Smed, H. Niinisalo, and H. Hakonen. "Realizing Bullet Time Effect in Multiplayer Games with Local Perception Filters". In: *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames '04. Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 121–128. ISBN: 158113942X. DOI: 10.1145/1016540.1016551. URL: https://doi.org/10.1145/1016540.1016551.

[Soc20] Socket.IO. *Socket.io. Realtime application framework (Node.JS server). Commit 2d2a31e*. 2020. URL: https://github.com/socketio/socket.io (visited on 10/15/2020).

[SRR98] P. M. Sharkey, M. D. Ryan, and D. J. Roberts. "A Local Perception Filter for Distributed Virtual Environments". In: *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium (Cat. No.98CB36180)*. Atlanta, Georgia, USA, 1998, pp. 242–249.

[SS15] J. Saldana and M. Suznjevic. “QoE and Latency Issues in Networked Games”. In: *Handbook of Digital Games and Entertainment Technologies*. Ed. by R. Nakatsu and M. Rauterberg. Singapore: Springer Singapore, 2015, pp. 1–36. ISBN: 9789814560528. DOI: 10.1007/978-981-4560-52-8_23-1. URL: https://doi.org/10.1007/978-981-4560-52-8_23-1.

[Sta13] A. R. Stagner. *Unity Multiplayer Games*. Community experience distilled. Birmingham: Packt Publishing, 2013. ISBN: 1849692335.

[Ste07] R. R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. Sept. 2007. DOI: 10.17487/RFC4960. URL: https://rfc-editor.org/rfc/rfc4960.txt.

[Ste97] W. R. Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001. Jan. 1997. DOI: 10.17487/RFC2001. URL: https://rfc-editor.org/rfc/rfc2001.txt.

[SV95] M. Shreedhar and G. Varghese. “Efficient Fair Queueing Using Deficit Round Robin”. In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’95. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1995, pp. 231–242. ISBN: 0897917111. DOI: 10.1145/217382.217453. URL: https://doi.org/10.1145/217382.217453.

[SZ03] K. Salen and E. Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003. ISBN: 0262240459.

[TBN06] J. Tulip, J. Bekkema, and K. Nesbitt. “Multi-Threaded Game Engine Design”. In: *Proceedings of the 3rd Australasian Conference on Interactive Entertainment*. IE ’06. Perth, Western Australia, Australia: Murdoch University, 2006, pp. 9–14. ISBN: 869059025.

[Tec15] Techopedia. *Network Performance*. 2015. URL: https://www.techopedia.com/definition/30022/network-performance (visited on 10/15/2020).

[TES14] H. Tschofenig, L. Eggert, and Z. Sarker. *Report from the IAB/IRTF Workshop on Congestion Control for Interactive Real-Time Communication*. RFC 7295. July 2014. DOI: 10.17487/RFC7295. URL: https://rfc-editor.org/rfc/rfc7295.txt.

[Tur86] J. Turner. “New Directions in Communications (Or Which Way to the Information Age?)” In: *IEEE Communications Magazine* 24.10 (1986), pp. 8–15.

[Tüx+15] M. Tüxen, R. Seggelmann, R. R. Stewart, and S. Loreto. *Additional Policies for the Partially Reliable Stream Control Transmission Protocol Extension*. RFC 7496. Apr. 2015. DOI: 10.17487/RFC7496. URL: https://rfc-editor.org/rfc/rfc7496.txt.

[Tüx+17] M. Tüxen, R. R. Stewart, R. Jesup, and S. Loreto. *Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets*. RFC 8261. Nov. 2017. DOI: 10.17487/RFC8261. URL: https://rfc-editor.org/rfc/rfc8261.txt.

[TW10] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks*. 5th. USA: Prentice Hall Press, 2010. ISBN: 0132126958.

[Vin18] Vince. *The Many Different Types of Video Games & Their Subgenres*. 2018. URL: https://www.idtech.com/blog/different-types-of-video-game-genres (visited on 10/15/2020).

[W3C20] W3C. *Web Real-Time Communications Working Group*. 2020. URL: https://www.w3.org/groups/wg/webrtc (visited on 10/15/2020).

[Web19] K. Webb. *The $120 billion gaming industry is going through more change than it ever has before, and everyone is trying to cash in*. 2019. URL: https://www.businessinsider.com/video-game-industry-120-billion-future-innovation-2019-9?r=US&IR=T (visited on 10/15/2020).

[Wei+99] W. Weiss, D. J. Heinanen, F. Baker, and J. T. Wroclawski. *Assured Forwarding PHB Group*. RFC 2597. June 1999. DOI: 10.17487/RFC2597. URL: https://rfc-editor.org/rfc/rfc2597.txt.

[WHA20] WHATWG. *The WebSocket interface, #dom-websocket-binarytype. HTML Living Standard — Last Updated 14 October 2020*. 2020. URL: https://html.spec.whatwg.org/multipage/web-sockets.html#dom-websocket-binarytype (visited on 10/15/2020).

[Wij19] T. Wijman. *The Global Games Market Will Generate $152.1 Billion in 2019 as the U.S. Overtakes China as the Biggest Market*. 2019. URL: https://newzoo.com/insights/articles/the-global-games-market-will-generate-152-1-billion-in-2019-as-the-u-s-overtakes-china-as-the-biggest-market/ (visited on 10/15/2020).

[WNG11] M. Welzl, F. Niederbacher, and S. Gjessing. “Beneficial Transparent Deployment of SCTP: The Missing Pieces”. In: *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*. Houston, Texas, USA, 2011, pp. 1–5.

[Won20] WonderNetwork. *Global Ping Statistics*. WonderNetwork. 2020. URL: https://wondernetwork.com/pings (visited on 10/15/2020).

[XI15] S. Xiong and H. Iida. “Attractiveness of Real Time Strategy Games”. In: *Computer Science and Information Systems* 12.4 (Nov. 2015), pp. 1217–1234. ISSN: 2406-1018. DOI: 10.2298/CSIS141101054X.

[XW13] J. Xu and B. W. Wah. “Concealing Network Delays in Delay-Sensitive Online Interactive GamesBbased on Just-Noticeable Differences”. In: *2013 IEEE International Conference on Multimedia and Expo (ICME)*. San Jose, California, USA, 2013, pp. 1–6.